UNIVERSITAT POLITÈCNICA DE CATALUNYA

PROYECTOS DE PROGRAMACIÓN
DISEÑO DE UN TECLADO

Q1 23/24
GRUPO 23.2

Guillem Angulo Hidalgo
guillem.angulo
Joan Martínez Soria
joan.martinez.soria
Tahir Muhammad Aziz
tahir.muhammad

VERSIÓN 3.0

ÍNDICE

1. DIAGRAMA DE CASOS DE USO....3
2. DESCRIPCIÓN DE LOS CASOS DE USO....4
2.1. Crear teclado....4
2.2. Poner nombre al teclado....4
2.3. Crear lenguaje....5
2.3.1. Crear alfabeto....5
2.3.2. Crear texto de referencia....5
2.3.2.1. Crear palabra....6
2.4. Editar un teclado....6
2.4.1. Cambiar nombre del teclado....7
2.4.2. Editar la posición de las teclas....7
2.4.3. Modificar lenguaje....7
2.4.3.1. Añadir texto....8
2.4.3.2. Eliminar texto....8
2.4.4. Borrar teclado....9
2.5. Buscar teclado....9
2.6. Ordenar teclados....9
3. DIAGRAMA DE CLASES....10
4. DESCRIPCIÓN DEL DIAGRAMA DE CLASES DEL DOMINIO....11
4.1. Teclado....11
4.2. Hungarian....11
4.3. AlgoritmoDisposicion....12
4.4. Lenguaje....12
4.5. Texto....13
4.5.1. TextoEstandar....13
4.5.2. TextoFrecuencia....14
4.6. Palabra....14
4.7. PalabraEnTexto....15
4.8. Alfabeto....15
4.9. Tecla....15
4.10. CtrLenguaje....16
4.11. CtrTeclado....16
4.12. CtrDomain....16
5. DESCRIPCIÓN DEL DIAGRAMA DE CLASES DE LA PERSISTENCIA....18
5.1. PersistenceTeclado....18
5.2. PersistenciaLenguaje....18
5.3. PersistenceAlfabeto....18
5.4. CtrPersistence....19
6. DESCRIPCIÓN DEL DIAGRAMA DE CLASES DE LA PRESENTACIÓN....20
6.1. ViewMenuPrincipal....20
6.2. ViewCrearTeclado....20
6.3. ViewEditarTeclado....20

6.4. ViewVisualizarTeclado....21
6.5. CtrPresentacion....21
7. Estructuras de Datos y Algoritmos....23
7.1. Dimensiones del teclado y su estrucutra de datos principal....23
7.2. AlgoritmoDisposicion....25
7.2.1. Introducción y estructuras de datos generales de la clase....25
7.2.2. Matrices de distancias y frecuencias....26
7.2.3. Branch and Bound....27
7.2.4. Cálculo de la cota....28
7.2.4.1. Cálculo del primer término....29
7.2.4.2. Cálculo del segundo término....29
7.2.4.2.1. Cálculo de la matriz C1....30
7.2.4.2.2. Cálculo de la matriz C2....31
7.2.4.2.3. Mejor asignación de C1C2 mediante Hungarian....32
7.2.5. Resultados obtenidos....34
8. Relación de clases implementadas....35
BIBLIOGRAFÍA....35

1. DIAGRAMA DE CASOS DE USO

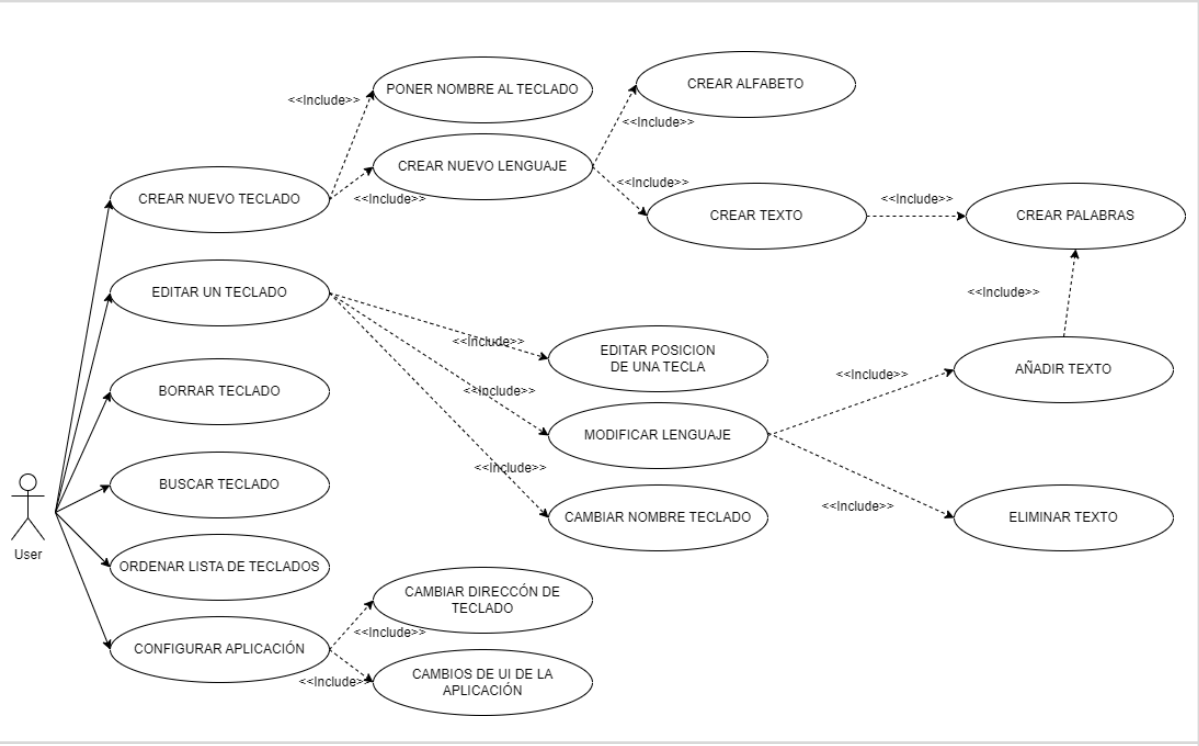

Figura 1. Diagrama de casos de uso

2. DESCRIPCIÓN DE LOS CASOS DE USO

2.1. Crear teclado

Nombre de la caso de uso	Crear teclado
Actor	Usuario
Precondición	-
Comportamiento	1. El usuario desea crear un teclado. 2. El usuario pone un nombre al teclado que está creando en ese momento (<<include>> Poner nombre teclado). 3. El usuario introduce los datos de un alfabeto (<<include>> Crear Lenguaje y Crear Alfabeto). 4. El usuario introduce los datos de un texto (<<include>> Crear Lenguaje y Crear Texto). 5. El usuario confirma la creación del teclado. 6. Se crea el nuevo lenguaje, el cual tiene asignados el alfabeto, el texto y las palabras pertenecientes a dicho texto (<<include>> Crear Lenguaje, Crear Alfabeto, Crear Texto y Crear Palabras). 7. El sistema, a partir de los datos anteriores, crea un nuevo teclado y asigna la posición de las teclas usando el algoritmo QAP para que este esté optimizado.
Errores posibles & casos alternativos	- Si el usuario cancela la creación, el proceso se detiene. - 5a. Si hay un error de formato al guardar la configuración, se muestra un mensaje de error dependiendo de la causa de este.

2.2. Poner nombre al teclado

Nombre de la caso de uso	Poner nombre al teclado
Actor	Usuario
Precondición	El usuario debe de estar creando un teclado.
Comportamiento	1. El usuario pone un nombre al teclado que está creando en ese momento. 2. El sistema, al confirmar la creación de dicho teclado, actualizará su información con ese mismo nombre.
Errores posibles & casos alternativos	- Si el usuario no pone nombre al teclado, se le pondrá uno por defecto (Nuevo Teclado).

2.3. Crear lenguaje

Nombre de la caso de uso	Crear lenguaje
Actor	Usuario
Precondición	El usuario debe de estar creando un teclado
Comportamiento	1. El usuario introduce los datos de un alfabeto (<<include>> Crear Alfabeto). 2. El usuario introduce los datos de un texto (<<include>> Crear Texto y Crear Palabras). 3. El sistema crea el lenguaje, con ese alfabeto y es texto y las palabras que pertenecen a dicho texto.
Errores posibles & casos alternativos	- Si hay algún error de formato a la hora de crear un alfabeto, se retornará un error. - Si se produce algún error de formato a la hora de crear un texto, se retornará un error.

2.3.1. Crear alfabeto

Nombre de la caso de uso	Crear alfabeto
Actor	Usuario
Precondición	El usuario debe de estar creando un teclado.
Comportamiento	1. El usuario introduce los datos del alfabeto (nombre y símbolos). 2. El sistema crea un alfabeto con los símbolos correspondientes.

Errores posibles & casos alternativos	- Si el usuario lo desea, puede introducir el mismo el teclado manualmente (tecleando el mismo el alfabeto dentro de un campo de texto, cada letra separada por un espacio) o vía fichero.
	  - Si el fichero proporcionado no es un fichero de texto, se mostrará un mensaje de error indicando lo sucedido.
	  - Si el fichero proporcionado no tiene símbolos se mostrará un mensaje de error indicando lo sucedido.

2.3.2. Crear texto de referencia

Nombre de la caso de uso	Crear texto de referencia
Actor	Usuario
Precondición	El usuario debe de estar creando un teclado.

Comportamiento	1. El usuario introduce manualmente un texto de referencia para crear el teclado (nombre y contenido del texto) (<<include>> Crear Palabra). 2. Al confirmar la creación de dicho teclado, el sistema creará ese texto.

Errores posibles & casos alternativos	- Si el usuario lo desea, puede introducir el mismo el teclado manualmente (tecleando el mismo el alfabeto dentro de un campo de texto, cada letra separada por un espacio) o vía fichero.
	  - Si el fichero proporcionado no es un fichero de texto, se mostrará un mensaje de error indicando lo sucedido.
	  - Si el fichero no contiene ninguna palabra, se mostrará un mensaje de error indicando lo sucedido.
	  - Si el fichero contiene una palabra que tiene algún símbolo que no pertenezca al alfabeto de su lenguaje, se retorna error que indicará lo que ha sucedido.

2.3.2.1. Crear palabra

Nombre de la caso de uso	Crear Palabra
Actor	Usuario
Precondición	El usuario debe de estar creando un teclado.
Comportamiento	1. El usuario confirma la creación de un teclado. 2. El sistema, a partir del texto introducido, genera las palabras que pertenecen al lenguaje.
Errores posibles & casos alternativos	- Si hay alguna palabra que no contenga uno de los símbolos del alfabeto, se retornará un mensaje de error.

2.4. Editar un teclado

Nombre de la caso de uso	Editar un teclado
Actor	Usuario
Precondición	El teclado debe existir.
Comportamiento	1. El usuario se encuentra probando un teclado y lo quiere modificar. 2. El usuario realiza todas aquellas modificaciones que quiera hacer respecto al teclado (nombre, distribución de las teclas o el texto de referencia) (<<include>> Cambiar nombre Teclado, Editar posición de las Teclas, Modificar Lenguaje). 3. El sistema actualiza los datos del teclado .

Errores posibles & casos alternativos	- Si el usuario cancela la edición, el proceso se detiene y el teclado se queda con los datos que tenía anteriormente. - Si se produce cualquier error de formato a la hora de modificar un teclado se mostrará un mensaje de error y no se realizará dicho cambio - Si alguna de las modificaciones del teclado implica la modificación del lenguaje, el teclado distribuirá las teclas nuevamente usando el algoritmo QAP

2.4.1. Cambiar nombre del teclado

Nombre de la caso de uso	Cambiar nombre del teclado
Actor	Usuario
Precondición	El teclado existe y el usuario lo está modificando.
Comportamiento	1. El usuario introduce un nuevo nombre para el teclado. 2. El sistema actualiza los datos del teclado.
Errores posibles & casos alternativos	- Si el nombre introducido es vacío, el nombre será el que tenía anteriormente.

2.4.2. Editar la posición de las teclas

Nombre de la caso de uso	Editar la posición de las teclas
Actor	Usuario
Precondición	El teclado existe y el usuario lo está modificando.
Comportamiento	1. El usuario realiza los cambios que necesite por lo que hace a la posición de las teclas. 2. El sistema actualiza los datos del teclado con las nuevas modificaciones.
Errores posibles & casos alternativos	- Si hay solapamiento de teclas o teclas repetidas se muestra un mensaje de error y el teclado se queda como estaba antes de iniciar dicha modificación.

2.4.3. Modificar lenguaje

Nombre de la caso de uso	Modificar Lenguaje
Actor	Usuario
Precondición	El teclado existe y el usuario lo está modificando.

Comportamiento	1. El usuario realiza las modificaciones que quiere hacer sobre el Lenguaje sobre el cual se ha construido el teclado (<<include>> Añadir Texto, Eliminar texto). 2. El sistema actualiza los datos y se hace una nueva distribución de las teclas teniendo en cuenta el nuevo texto usando el algoritmo QAP.

Errores posibles & casos alternativos	- El usuario introduce el texto de referencia actualizado vía fichero de texto.
	  - Si el fichero proporcionado no es un fichero de texto, se mostrará un mensaje de error y no se realizará ningún cambio.
	  - Si el texto contiene alguna palabra con símbolos que no pertenecen al alfabeto del lenguaje, se retorna un mensaje de error.

2.4.3.1. Añadir texto

Nombre de la caso de uso	Añadir texto
Actor	Usuario
Precondición	El usuario está modificando un teclado.
Comportamiento	1. El usuario introduce los datos del nuevo texto. 2. El sistema crea y asigna las nuevas Palabras que no existían previamente al Lenguaje que se está modificando (<<include>> Crear Palabra). 3. El sistema actualiza los datos y se hace una nueva distribución de las teclas teniendo en cuenta el nuevo texto usando el algoritmo QAP.

Errores posibles & casos alternativos	- El usuario introduce el texto de referencia actualizado vía fichero de texto.
	  - Si el fichero proporcionado no es un fichero de texto, se mostrará un mensaje de error y no se realizará ningún cambio.
	  - Si el texto contiene alguna palabra con símbolos que no pertenecen al alfabeto del lenguaje, se retorna un mensaje de error.

2.4.3.2. Eliminar texto

Nombre de la caso de uso	Eliminar texto
Actor	Usuario
Precondición	Se está modificando el lenguaje de un teclado.
Comportamiento	1. El usuario selecciona el texto que quiere eliminar del lenguaje.

	2. El sistema actualiza los datos y se hace una nueva distribución de las teclas teniendo en cuenta el cambio, usando el algoritmo QAP.
Errores posibles & casos alternativos	- Si el lenguaje solo tiene un texto, muestra un mensaje de que no se pueden eliminar teclados.

2.4.4. Borrar teclado

Nombre de la caso de uso	Borrar teclado
Actor	Usuario
Precondición	El teclado existe.
Comportamiento	1. El usuario quiere borrar uno de los teclados. 2. El usuario confirma la eliminación del teclado. 3. El teclado es borrado del sistema.
Errores posibles & casos alternativos	- Si el usuario cancela la acción el teclado no es eliminado.

2.5. Buscar teclado

Nombre de la caso de uso	Buscar teclado
Actor	Usuario
Precondición	-
Comportamiento	1. El usuario quiere buscar un teclado concreto. 2. El usuario introduce el nombre o palabra clave del teclado que quiera buscar. 3. El sistema busca aquellos teclados que tengan parte del nombre introducido por el usuario. 4. El sistema muestra la lista de teclados que cumplen la condición.
Errores posibles & casos alternativos	- Si no hay ningún teclado que cumpla la condición(no hay ningún teclado que tenga esa combinación de caracteres), se mostrará un mensaje expresando lo sucedido

2.6. Ordenar teclados

Nombre de la caso de uso	Ordenar teclados
Actor	Usuario

Precondición	-
Comportamiento	1. El usuario selecciona como quiere ordenar la lista de teclados. 2. El usuario confirma su selección. 3. El sistema reordena la lista de teclados según lo haya escogido el usuario (fecha de creación, modificación o nombre, ya sea de manera ascendente o descendente). 4. El sistema muestra los teclados ordenados como el usuario ha indicado.
Errores posibles & casos alternativos	-

3. DIAGRAMA DE CLASES

Dado que el diagrama cubre las tres capas, su resolución es muy grande como para adjuntarlo aquí como en el caso del diagrama de casos. Por ello, para ver el diagrama de clases actualizado, se puede encontrar en formato png en los archivos de la entrega.

4. DESCRIPCIÓN DEL DIAGRAMA DE CLASES DEL DOMINIO

4.1. Teclado

Nombre de la clase	Teclado
Breve descripción	Teclado formado por teclas para un lenguaje en concreto.
Cardinalidad	Un teclado puede ser el más óptimo para un lenguaje, y un teclado está formado por una o más teclas, la lógica de los teclados es controlada por el CtrTeclados.
Descripción de los atributos	- **nombre**: Nombre puesto por el usuario al teclado. - **numTeclas**: Número de - **numFilas**: Número de filas totales que contiene el teclado. - **numColumnas**: Número de columnas totales que contiene el teclado. - **tamUltimaFila:** Representa el tamaño de la última fila. - **offset**: Tanto el anterior atributo como este se explican detalladamente en el apartado de ED y A. - **fechaCreacion**: Indica el día en que se realizó la creación del teclado. - **fechaModificacion**: Indica cuál fue la fecha en la que se modifica el teclado por última vez, en caso de que no se haya modificado, es la misma fecha en la que se creó el teclado.
Descripción de las relaciones	- **Relación de asociación con "Lenguaje"**: Indica para qué lenguaje o lenguajes este teclado es el mejor. - **Relación de composición con "Tecla"**: Indica por qué teclas está compuesto el teclado. - **Relación de asociación con "AlgoritmoDisposición"**: Indica qué algoritmo se ha usado para generar el teclado. - **Relación de agregación con "CtrTeclado"**: Indica que el CtrTeclado está formado por ninguno, o muchos teclados.

4.2. Hungarian

Nombre de la clase	Hungarian
Breve descripción	Clase que se encarga de la ejecución del Hungarian Algorithm.
Cardinalidad	La clase Hungarian es usada por una clase AlgoritmoDisposicion para realizar sus cálculos.
Descripción de los atributos	- **matriz**: Es la matriz sobre la que se aplica el problema de extraer la asignación óptima. - **cuadroEnFila**: Arreglo de enteros que indica la posición de los ceros marcados en cada fila.

	- **cuadroEnColumna**: Un arreglo de enteros que indica la posición de los ceros marcados en cada columna. - **filaEstaCubierta**: Indica si una fila está cubierta, se usa para indicar qué filas son parte de la asignación óptima. - **columnaEstaCubierta**: Análogo al anterior con columnas. - **cerosMarcadosEnFila**: Almacena la posición de los ceros marcados en cada fila.
Descripción de las relaciones	**Relación de asociación con "Teclado"**: Indica para qué teclado es ese algoritmo.

4.3. AlgoritmoDisposicion

Nombre de la clase	AlgoritmoDisposicion
Breve descripción	Algoritmo encargado de generar el mejor teclado posible.
Cardinalidad	Un algoritmo se aplica en uno o más teclados, y todo teclado tiene un algoritmo que lo ha optimizado.
Descripción de los atributos	- **NUM_OBJETOS:** número de instalaciones y ubicaciones sobre las que hay que aplicar el algoritmo. - **matrizDistancias**: matriz que representa la distancia entre cada par de ubicaciones. - **matrizFrecuencias**: matriz que almacena la frecuencia entre cada par de instalaciones. - **mejorCota**: guarda el valor de la mejor cota obtenida al aplicar el algoritmo. - **mejorSolucion**: considerando que representa la mejor solución, es un vector que guarda la jesima ubicación que le corresponde al iesimo símbolo, es decir mejorSolucion[iesimoSimbolo] = jesimaUbicacion - **podas**: guarda el número de podas que se han hecho en cada nivel del árbol de soluciones, su uso es para medir la efectividad durante la realización o mantenimiento del algoritmo. Se detalla mejor en el apartado de estructuras de datos y algoritmos.
Descripción de las relaciones	- **Relación de asociación con "Teclado"**: Indica para qué teclado es ese algoritmo.

4.4. Lenguaje

Nombre de la clase	Lenguaje
Breve descripción	Lenguaje formado por un alfabeto concreto y uno o más textos de diferentes tipos que contienen palabras de dicho alfabeto.
Cardinalidad	Un lenguaje está basado en un alfabeto, está formado por uno o más textos y está compuesto por un conjunto de palabras que pertenecen a

	sus textos y estas se forman a partir del alfabeto del lenguaje. Hay un solo teclado óptimo para ese lenguaje. La lógica de los Lenguajes es controlada por el CtrLenguaje.
Descripción de los atributos	- **nombre**: Nombre dado al lenguaje. - **frecuencias**: Matriz que almacena la frecuencia entre los diferentes pares de símbolos del alfabeto. - **update**: Booleano que indica si el lenguaje ha sido actualizado, ya sea mediante la adición o eliminación de un texto.
Descripción de las relaciones	- **Relación de asociación con "Teclado"**: Indica qué teclado es el mejor para ese lenguaje. - **Relación de asociación con "Texto"**: Indica qué textos han sido creados y dados en ese lenguaje. - **Relación de asociación con "Alfabeto"**: Indica mediante qué alfabeto se ha creado ese lenguaje. - **Relación de composición con "Palabra"**: Indica el conjunto de palabras que forman el lenguaje, y tienen que ser palabras que aparezcan en sus textos. - **Relación de agregación con "CtrLenguaje"**: Indica que el CtrLenguaje está formado por ninguno, o muchos lenguajes.

4.5. Texto

Nombre de la clase	Texto
Breve descripción	Conjunto de palabras que forman un texto.
Cardinalidad	Un texto está formado por una o más palabras, y éste pertenece a uno o a distintos lenguajes.
Descripción de los atributos	- **nombre**: título identificativo de cada texto. - **texto**: contenido del texto en cuestión. - **type**: tipo de texto
Descripción de las relaciones	- **Relación de asociación con "Lenguaje"**: Indica en qué lenguaje ha sido introducido el texto. - **Relación de asociación con "PalabraEnTexto"**: Indica qué palabras forman el texto.

4.5.1. TextoEstandar

Nombre de la clase	TextoEstandar
Breve descripción	Representa un texto regular que consta de palabras y signos de puntuación. Es una de las subclases de la clase "Texto".
Cardinalidad	Cada instancia de TextoNormal está asociada a un único texto y hereda las propiedades y métodos de la clase Texto.

Descripción de los atributos	- **nombre**: título identificativo de cada texto. - **texto**: se trata del contenido del texto. - **/type**: Tipo de texto al que pertenece, siempre debe ser "Normal".
Descripción de las relaciones	-

4.5.2. TextoFrecuencia

Nombre de la clase	TextoFrecuencia
Breve descripción	Representa un texto, donde cada palabra va siempre acompañada de la cantidad de veces que aparece en el texto, es decir, su frecuencia. Ejemplo: Entropía 20 Quantum 30.
Cardinalidad	Cada instancia de TextoFrecuencia está asociada a un único texto y hereda las propiedades y métodos de la clase Texto.
Descripción de los atributos	- **titulo**: título identificativo de cada texto. - **texto**: se trata del contenido del texto. - **/type**: Tipo de texto al que pertenece, siempre debe ser "frecuencia".
Descripción de las relaciones	-

4.6. Palabra

Nombre de la clase	Palabra
Breve descripción	Conjunto de símbolos incluidos en un alfabeto que forman una palabra.
Cardinalidad	Una palabra pertenece a un alfabeto y cuenta con símbolos de ese alfabeto. Toda palabra que aparezca en un texto de un lenguaje también formará parte de este.
Descripción de los atributos	- **nombre**: la palabra en sí.
Descripción de las relaciones	- **Relación de asociación con "PalabraEnTexto"**: Indica en qué texto aparece (si aparece) esa palabra. - **Relación de agregación con "Lenguaje"**: Indica en qué lenguaje forma parte esa palabra. - **Relación de asociación con "Alfabeto"**: Indica mediante qué alfabeto se ha creado la palabra.

4.7. PalabraEnTexto

Nombre de la clase	PalabraEnTexto
Breve descripción	Palabra y número de veces que aparece ésta en un texto.
Cardinalidad	Una palabra aparece un número de veces en un texto.
Descripción de los atributos	- **numApariciones**: Número de veces que aparece esa palabra en un texto.
Descripción de las relaciones	- **Relación de asociación con "Palabra"**: Indica qué palabra aparece en un texto. - **Relación de asociación con "Texto"**: Indica en qué texto aparece la palabra.

4.8. Alfabeto

Nombre de la clase	Alfabeto
Breve descripción	Conjunto de símbolos que forman un alfabeto.
Cardinalidad	Un alfabeto puede formar uno o más lenguajes.
Descripción de los atributos	- **nombre**: Nombre del alfabeto. - **simbolos**: Los símbolos por los que se compone.
Descripción de las relaciones	- **Relación de asociación con "Palabra"**: Indica que la palabra está construida con ese alfabeto. - **Relación de asociación con "Lenguaje"**: Indica si hay algún o algunos lenguajes basados en ese alfabeto.

4.9. Tecla

Nombre de la clase	Tecla
Breve descripción	Símbolo representado en una posición concreta del teclado
Cardinalidad	Una tecla está formada por un único símbolo.
Descripción de los atributos	- **simbolo**: Símbolo que representa dicha tecla.
Descripción de las relaciones	- **Relación de composición con "Teclado"**: Indica a qué teclado pertenece esa tecla.

4.10. CtrLenguaje

Nombre de la clase	CtrLenguaje
Breve descripción	Contiene toda la lógica del modelo por lo que hace a la gestión de los diferentes Lenguajes.
Cardinalidad	Esta clase contiene un conjunto de Lenguajes y esta una instancia de esta clase está contenida dentro del CtrDomain.
Descripción de los atributos	-
Descripción de las relaciones	- **Relación de agregación con "Lenguaje"**: Indica que el CtrLenguaje está formado por ninguno, o muchos lenguajes. - **Relación de asociación con "CtrDomain"**: Indica que el CtrDomain contiene una instancia de CtrLenguaje.

4.11. CtrTeclado

Nombre de la clase	CtrTeclado
Breve descripción	Contiene toda la lógica del modelo por lo que hace a la gestión de los diferentes Teclados.
Cardinalidad	Esta clase contiene un conjunto de Teclados y esta una instancia de esta clase está contenida dentro del CtrDomain.
Descripción de los atributos	-
Descripción de las relaciones	- **Relación de agregación con "Teclado"**: Indica que el CtrTeclado está formado por ninguno, o muchos teclados. - **Relación de asociación con "CtrDomain"**: Indica que el CtrDomain contiene una instancia de CtrTeclado.

4.12. CtrDomain

Nombre de la clase	CtrDomain
Breve descripción	Contiene toda la lógica de negocio de todo el dominio de nuestra aplicación, se trata de una clase Singleton.
Cardinalidad	Esta clase contiene una instancia del CtrTeclado, CtrLenguaje y CtrPersistence, y a su vez está contenida en el CtrPresentacion para que esta pueda realizar las diferentes llamadas al dominio.
Descripción de los	-

atributos	
Descripción de las relaciones	- **Relación de asociación con "CtrTeclado"**: Indica que el CtrDomain contiene una instancia de CtrTeclado. - **Relación de asociación con "CtrLenguaje"**: Indica que el CtrDomain contiene una instancia de CtrLenguaje. - **Relación de asociación con "CtrPersistence"**: Indica que el CtrDomain contiene una instancia de CtrPersistence. - **Relación de asociación con "CtrPresentation"**: Indica que el CtrPresentation contiene una instancia de CtrDomain.

5. DESCRIPCIÓN DEL DIAGRAMA DE CLASES DE LA PERSISTENCIA

5.1. PersistenceTeclado

Nombre de la clase	PersistenceTeclado
Breve descripción	Clase que obtiene, guarda y modifica los Teclados creados por el usuario en los recursos del programa para ser utilizados siempre que se necesiten. Se trata de una clase Singleton.
Cardinalidad	El CtrPersistence contiene una única instancia de PersistenceTeclado
Descripción de los atributos	-
Descripción de las relaciones	- **Relación de asociación con "CtrPersistence"**: Indica que el CtrPersistence contiene una instancia de PersistenceTeclado.

5.2. PersistenciaLenguaje

Nombre de la clase	PersistenceLenguaje
Breve descripción	Clase que obtiene, guarda y modifica los Lenguajes creados por el usuario en los recursos del programa para ser utilizados siempre que se necesiten. Se trata de una clase Singleton.
Cardinalidad	CtrPersistence contiene una única instancia de PersistenceLengauje.
Descripción de los atributos	-
Descripción de las relaciones	- **Relación de asociación con "CtrPersistence"**: Indica que el CtrPersistence contiene una instancia de PersistenceLenguaje.

5.3. PersistenceAlfabeto

Nombre de la clase	PersistenceAlfabeto
Breve descripción	Clase que obtiene los Alfabetos almacenados en los recursos del programa para poder ser usados como alfabetos ya hechos y que el usuario puede usar a la hora de crear un teclado. Se trata de una clase Singleton.
Cardinalidad	CtrPersistence contiene una única instancia de PersistenceAlfabeto.

Descripción de los atributos	-
Descripción de las relaciones	- **Relación de asociación con "CtrPersistence"**: Indica que el CtrPersistence contiene una instancia de PersistenceAlfabeto.

5.4. CtrPersistence

Nombre de la clase	CtrPersistence
Breve descripción	Clase que contiene toda la lógica de la persistencia y que se comunica con el dominio por tal de realizar las diferentes actividades necesarias para almacenar y modificar los diferentes aspectos de nuestra aplicación. Se trata de una clase Singleton.
Cardinalidad	Esta clase contiene una instancia de PersistenceTeclado, PersistenceLenguaje y PersistenceAlfabeto, y esta se comunica con el CtrDomain.
Descripción de los atributos	-
Descripción de las relaciones	- **Relación de asociación con "CtrDomain"**: Indica que el CtrDomain contiene una instancia de CtrPersistence. - **Relación de asociación con "PersistenceTeclado"**: Indica que el CtrPersistence contiene una instancia de PersistenceAlfabeto. - **Relación de asociación con "PersistenceLenguaje"**: Indica que el CtrPersistence contiene una instancia de PersistenceAlfabeto. - **Relación de asociación con "PersistenceAlfabeto"**: Indica que el CtrPersistence contiene una instancia de PersistenceAlfabeto.

6. DESCRIPCIÓN DEL DIAGRAMA DE CLASES DE LA PRESENTACIÓN

6.1. ViewMenuPrincipal

Nombre de la clase	ViewMenuPrincipal
Breve descripción	Esta es la clase encargada de mostrar el menú principal de nuestra aplicación, donde se mostraran los diferentes teclados generados por el usuario, los podrá buscar por su nombre y los podrá ordenar.
Cardinalidad	Esta clase tendrá una relación con la clase CtrPresentation, que será la encargada de decirle cuando tiene que aparecer.
Descripción de los atributos	Los atributos que contiene esta clase son los diferentes widgets necesarios para poder crear la pantalla y poder interactuar con ella.
Descripción de las relaciones	- **Relación de asociación con "CtrPresentation"**: Indica que el CtrPresentation está comunicado con la pantalla principal y pueden realizar llamadas mutuas.

6.2. ViewCrearTeclado

Nombre de la clase	ViewCrearTeclado
Breve descripción	Esta es la clase encargada de mostrar la vista para crear un teclado, en esta encontramos un formulario que el usuario deberá de rellenar con toda la información necesaria para poder crear un teclado optimizado.
Cardinalidad	Esta clase tendrá una relación con la clase CtrPresentation, que será la encargada de decirle cuando tiene que aparecer.
Descripción de los atributos	Los atributos que contiene esta clase son los diferentes widgets necesarios para poder crear la pantalla y poder interactuar con ella
Descripción de las relaciones	- **Relación de asociación con "CtrPresentation"**: Indica que el CtrPresentation está comunicado con la pantalla de creacion de un teclado y pueden realizar llamadas mutuas.

6.3. ViewEditarTeclado

Nombre de la clase	ViewEditarTeclado
Breve descripción	Esta es la clase encargada de mostrar el la vista encargada de la edición del teclado, en esta se mostraran las diferentes opciones de modificar un teclado.

Cardinalidad	Esta clase tendrá una relación con la clase CtrPresentation, que será la encargada de decirle cuando tiene que aparecer.
Descripción de los atributos	Los atributos que contiene esta clase son los diferentes widgets necesarios para poder crear la pantalla y poder interactuar con ella.
Descripción de las relaciones	- **Relación de asociación con "CtrPresentation"**: Indica que el CtrPresentation está comunicado con la pantalla de edición de un teclado concreto y pueden realizar llamadas mutuas.

6.4. ViewVisualizarTeclado

Nombre de la clase	ViewVisualizarTeclado
Breve descripción	Esta clase es la encargada de mostrar la vista donde el usuario podrá visualizar el teclado. Esta contendrá toda su información más relevante y es donde el usuario podrá probar el teclado.
Cardinalidad	Esta clase tendrá una relación con la clase CtrPresentation, que será la encargada de decirle cuando tiene que aparecer.
Descripción de los atributos	Los atributos que contiene esta clase son los diferentes widgets necesarios para poder crear la pantalla y poder interactuar con ella.
Descripción de las relaciones	- **Relación de asociación con "CtrPresentation"**: Indica que el CtrPresentation está comunicado con la pantalla para visualizar un teclado concreto.

6.5. CtrPresentacion

Nombre de la clase	CtrPresentation
Breve descripción	Esta es la clase encargada de controlar la lógica de las vistas de nuestra aplicación, por lo tanto, es la encargada de mostrar en todo momento la vista que se requiera en cualquier momento y de comunicarse con el CtrDomain para poder realizar los diferentes casos de uso.
Cardinalidad	Esta clase se comunicara con las diferenets vistas de nuestra aplicación y se comunicara con el CtrDomain.
Descripción de los atributos	
Descripción de las relaciones	- **Relación de asociación con "ViewMenuPrincipal"**: Indica que el CtrPresentation está comunicado con la pantalla principal y pueden realizar llamadas mutuas. - **Relación de asociación con "ViewCrearTeclado"**: Indica que el CtrPresentation está comunicado con la pantalla de creación de un teclado.

	- **Relación de asociación con “ViewEditarTeclado”**: Indica que el CtrPresentation está comunicado con la pantalla de edición de un teclado. - **Relación de asociación con “ViewVisualizarTeclado”**: Indica que el CtrPresentation está comunicado con la pantalla para visualizar un teclado concreto.

7. Estructuras de Datos y Algoritmos

En el núcleo de nuestro proyecto de diseño de teclados personalizados y distribuciones óptimas se encuentran las estructuras de datos y algoritmos que hemos creído que retienen tanto una buena eficiencia como la usabilidad del sistema. En esta sección, se detallarán las decisiones fundamentales tomadas para organizar y manipular la información clave, así como el algoritmo diseñado para lograr una disposición de teclas óptima dado un lenguaje específico.

7.1. Dimensiones del teclado y su estrucutra de datos principal

Para una buena distribución de teclas y así minimizar las distancias al teclear con un dedo un lenguaje, primero nos preguntamos las dimensiones que debería de tener el teclado. Llegamos a la conclusión de que un teclado cuadrado es una buena opción. Sin embargo, no todos los alfabetos sobre los que operaremos serán del mismo tamaño, y por ello no todos se podrán dimensionar de forma cuadrada perfecta.

Teniendo en cuenta lo anterior y sin dejar de lado la idea de un teclado cuadrado, la mejor opción sería intentar construir un teclado idealmente cuadrado que alcanzamos calculando la raíz cuadrada inferior para el tamaño del alfabeto. Si no se han cubierto todas las teclas, se debe añadir una fila y volver a comprobar si se han cubierto, en caso que tampoco, añadimos una columna y nos aseguramos de cubrirla siempre gracias a las propiedades matemáticas. Por ejemplo:

- Alfabeto de tamaño 5. Se calcularía la raíz cuadrada por debajo, que da 2. Como vemos que un teclado 2x2 no cubriría las 5 letras, ya que faltaría una, añadimos una fila, obteniendo así la dimensión 3x2 que cubre todas los símbolos.
- Alfabeto de tamaño 6: Siguiendo exactamente el mismo proceso que antes, acabamos obteniendo un teclado de dimensión 2x3. Con la diferencia de que es exacto y no sobran teclas, que tiene importancia en teclados grandes como veremos luego.
- Alfabeto de tamaño 7: Veamos que pasa si al añadir una fila, tampoco cubre los símbolos. Igual que en los anteriores casos, se empezaría por un 2x2, luego se incrementa a 3x2 y dado que sigue sin satisfacer el tamaño, se añadiría una columna para obtener así un teclado 3x3.

La implementación de esto es muy básica:

```java
int numTeclas = tamAlfabeto;
int numFilas = (int)raiz(numTeclas);
int numColumnas = numFilas;
if (numFilas*numColumnas < numTeclas)
        ++numFilas;
if (numFilas*numColumnas < numTeclas)
        ++numColumnas;
```

Pero como se ha destacado, con algunos tamaños quedan huecos y es importante ver qué casillas se dejan vacías. Lógicamente al ejecutar el código no vamos a comprobar todas las combinaciones de huecos que se van a dejar en el teclado porque sería extremadamente lento, por lo que necesitamos una opción general viable.

Pensamos que dejar los huecos al final de la última fila y centrar los que hayan es una buena alternativa. Centrar la última fila es importante, ya que si necesitamos construir teclados grandes, por ejemplo con un alfabeto de tamaño 26, obtendremos un teclado 6x5 donde la última fila tiene un único símbolo, y si lo dejamos en la tercera columna en vez de la primera, reducimos el caso peor ya que las distancias con la primera fila son menores. En la siguiente figura se puede ver mejor, a pesar de que es un concepto trivial:

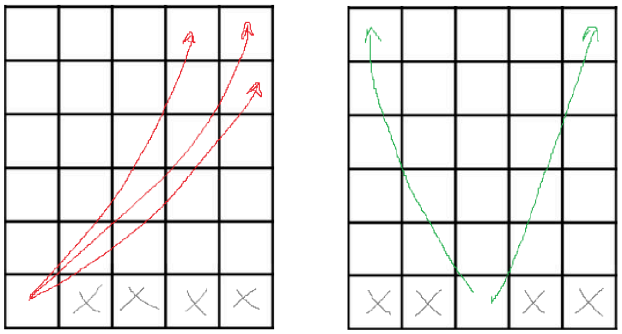

Figura 2. Croquis de la de las distancias dependiendo de la ubicación de las teclas sobrantes.

El anterior código se puede expandir con lo siguiente para obtener los números que necesitamos para representar el teclado de esa forma:

```Java
int espaciosVacios = numFilas*numColumnas - numTeclas;
int tamUltimaFila = numColumnas - espaciosVacios;
int offset = espaciosVacios / 2;      // representa la primera posicion valida en la
                                      // ultima fila
```

Primeramente, se eligió implementarlo con matrices ya que la propia naturaleza así lo sugiere, usando NULL en los huecos vacíos, sin embargo, durante las últimas fases de la implementación de todo el proyecto, pensamos que era mejor idea usar **listas de Tecla**. Esto era, entre otras cosas, por si en un futuro quisiéramos colocar las teclas en puntos (o coordenadas) más dispersas, como en los teclados habituales donde entre fila y fila hay un desplazamiento, y porque facilita la escritura del código.

Internamente se puede seguir viendo el teclado como matriz, pero realmente **trabajamos con i-ésimos en la lista, pero sabemos que cada tecla tiene una posición Point que se calcula únicamente cuando es necesario con la ayuda de numTeclas, numFilas y offset**.

7.2. AlgoritmoDisposicion

7.2.1. Introducción y estructuras de datos generales de la clase

La optimización que se nos presenta a realizar es la conocida como QAP (quadratic assignment problem, en inglés). Vamos a describir el problema brevemente y luego la implementación que hemos hecho y las estructuras de datos que se han usado.

El QAP presenta *n* ubicaciones y *n* instalaciones, para cada par de ubicación sabemos la distancia que las separa y para cada par de instalaciones el flujo (o coste) que hay entre ellos. Nuestra misión es emplazar las *n* instalaciones en cada ubicación diferente entre ellas para minimizar el coste total del sistema, es decir, minimizar:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} ubicación(Xi, Xj) * frecuencia(i, j)$$

Donde i y j son dos instalaciones, y Xi y Xj sus respectivas ubicaciones.

Es un problema NP-completo, por lo que para valores grandes, el coste de ejecución es cada vez mayor, pero mediante técnicas que describiremos a continuación hemos podido alcanzar una solución general casi perfecta con tamaños altos, y un tiempo de ejecución muy corto.

Antes de entrar en detalle del funcionamiento del algoritmo, vamos a describir los atributos que usa y el por qué de sus estructuras de datos.

```java
Teclado teclado;          // teclado del algoritmo, nos porporciona detalles sobre
                                  // su lenguaje, alfabeto, etc.
List<Character> simbolos; // alfabeto como lista de caracteres

final int NUM_OBJETOS;    // el numero de n de instalaciones y ubicaciones

int[][] matrizDistancias; // matriz de distancias entre ubicaciones Xi Xj
int[][] matrizFrecuencias;        // matriz de frecuencias de entre dos simbolos

int mejorCota;                    // guarda la mejor cota encontrada por el algoritmo
int[] mejorSolucion;   // guarda la jesima ubicacion que le corresponde al iesimo
                          // simbolo: mejorSolucion[iesimoSimbolo] = jesimaUbicacion
```

Los símbolos del alfabeto como lista de caracteres tiene que ver más como la implementación del proyecto que una preferencia sobre las otras posibles representaciones, sin embargo, no existe ninguna que nos sea útil más eficiente y cómoda teniendo en cuenta las operaciones que se hacen sobre ella. Cabe destacar también que se usa como atributo por comodidad, ya que realmente al tener teclado (por donde podemos obtener los símbolos directamente) no haría falta.

La elección de matrices para indicar las distancias entre cada par de ubicaciones, y las frecuencias entre cada par de instalaciones es más que obvia, la propia descripción nos sugiere usar matrices de tamaño *n* x *n*.

Respecto a la mejor cota, usamos un entero, y su funcionalidad exacta se detalla al describir el algoritmo en los próximos apartados.

El último atributo, que representa la mejor solución, se guarda como un array de tamaño *n*, en ella se encuentra la ubicación del símbolo, todo esto se hace en i-ésimos para no tener que depender de la implementación del teclado: incluso si la clase Teclado tuviera el teclado en forma de matriz pura, el algoritmo seguiría funcionando perfectamente, ya que no necesitamos saber los tamaños en ningún momento. Solo dependemos del teclado para saber las distancias entre dos ubicaciones, que puede ser cualquiera (lo que refuerza la idea de tener ubicaciones irregulares comentada anteriormente).

También disponemos de estructuras básicas que nos ayudan a realizar el algoritmo como las siguientes:

```java
// numero de simbolos colocados
    int numObjetosColocados = 0;
// indica si el simbolo i esta colocado
    boolean[] iesimoSimboloColocado = new boolean[NUM_OBJETOS];
// indica si la iesima ubicacion esta ocupada
    boolean[] iesimaUbicacionOcupada = new boolean[NUM_OBJETOS];
// indica la jesima ubicacion que le corresponde al iesimo simbolo, el valor retornado es valido
si iesimoSimboloColocado[i] == true
    int[] jesimaUbicacionDeIesimoSimbolo = new int[NUM_OBJETOS];
```

Finalmente, tenemos una clase dentro del propio algoritmo que se puede entender como un struct de C++; es decir, es un tipo de dato que usamos para el algoritmo al usar branch and bound y comparar soluciones. Su implementación es muy simple puesto que solo tiene los atributos, la constructora y los getters.

```java
static public class SolucionParical{
    private int cota;
    private int numObjetosColocados;
    private boolean[] iesimoSimboloColocado;
    private boolean[] iesimaUbicacionOcupada;
    private int[] jesimaUbicacionDeIesimoSimbolo;

// constructora normal que tiene como parametros todos los atributos
     ...
// getters de todos los atributos
     ...
}
```

7.2.2. Matrices de distancias y frecuencias

Ahora nos adentramos ya en el funcionamiento del algoritmo. Primeramente hay que obtener las matrices de distancias y frecuencias. Para ello nos apoyamos en la clase del teclado y la del lenguaje respectivamente, como es lógico, ya que uno depende del teclado (ubicaciones) y el otro del lenguaje (por sus textos).

7.2.3. Branch and Bound

Construir todas las permutaciones del problema, calcular su coste y elegir el del menor sería extremadamente lento hasta para valores de *n* pequeños. Entonces, para afrontar las combinaciones se ha usado el **enfoque Branch and Bound**. Este enfoque se respalda en el cálculo de una **cota** que nos dirá a medida que vamos construyendo una solución (es decir, que tenemos una solución parcial) qué potencial tiene de cara a la cualquier solución final que se pueda construir a partir de ahí. De esta forma, si encontramos que la cota pronostica una solución parcial peor que otra que tenemos hasta el momento, podemos descartarla y ahorrar mucho cómputo.

Se puede entender en forma de árbol, aunque no necesariamente tiene que estar implementado como tal. Para cada solución parcial se puede pensar como un nodo, el nodo padre es la solución parcial de la que deriva, es decir, respecto a su padre se ha tomado una decisión: colocar un símbolo en una ubicación del teclado. Y cada nodo tiene tantos hijos como soluciones parciales que se pueden conseguir determinando un símbolo que no ha sido colocado en una ubicación libre.

De esto se encarga la siguiente función (se ha recortado y cambiado para simplificar su comprensión):

Java

```
private void branchAndBoundv2(SolucionParical solucionParcial) {
     // si todos los simbolos estan colocados (teclado completo) se guarda, es la mejor sol
     if (numObjetosColocados == NUM_OBJETOS) {
         mejorCota = solucionParcial.getCota();
         mejorSolucion = solucionParcial.getSolucion();
     }

     // se crea una cola de prioridad de SolucionParical, ordenada por coste:
     PriorityQueue<SolucionParical> pq = new PriorityQueue<SolucionParical>(...);

     // se rellena la cola de prioridad con las soluciones parciales que se puedan construir
     for (int i = 0; i < NUM_OBJETOS; ++i) {
         if (!iesimoSimboloColocado[i]) {               // por cada simbolo no colocado
             iesimoSimboloColocado[i] = true;

             for (int j = 0; j < NUM_OBJETOS; ++j) {
                 if (!iesimaUbicacionOcupada[j]) {      // por cada ubicacion no ocupada
                     iesimaUbicacionOcupada[j] = true;

                     int cotaHijo = calcularCotaGilmoreLawler(...);  // se calcula la cota
                     pq.add(...);                                    // y se mete en la cola

                     iesimaUbicacionOcupada[j] = false;
                 }
             }

             iesimoSimboloColocado[i] = false;
         }
     }

     // se van sacando las soluciones parciales de la cola de prioridad y se llama
     // recursivamente a la funcion con ellas
     while (!pq.isEmpty()) {
         SolucionParical top = pq.poll();
         if (top.getCota() < mejorCota)     // NOTA1: && top.getCota() == Infinito
             branchAndBoundv2(top);
          }
 }
```

Se puede observar que como entrada recibimos una solución parcial descrita anteriormente en el apartado de las estructuras de datos, con ella miramos si es completa, si lo es, se actualiza.

Si no lo es, usamos una estructura de datos nueva, **colas de prioridad de soluciones parciales** que se ordenan mediante sus costes en forma ascendente, por lo que la primera será siempre la de menor coste, es decir, la que más potencial tiene.

Esta cola se rellena con el cálculo de todas las cotas posibles de todos sus hijos. Una vez rellenada, hacemos un bucle que consiste en vaciar la cola, comprobando si el elemento de más adelante es menor a la mejor solución hasta ahora. Teniendo en cuenta que se inicializa a infinito, si aún no hay solución, se hace branching de la cabeza que hay en la cola (la aparentemente mejor solución) hasta llegar a la final. Esto es importante porque nos permite llegar a una solución de forma muy rápida: bajamos exactamente *n* (siendo *n* el número de ubicaciones e instalaciones) niveles del árbol, ya que cada nivel tiene su propia cola de prioridad. Se puede llegar a pensar en ella como una solución voraz.

Ahora se va a comentar sobre una decisión que se ha tomado teniendo en cuenta el rendimiento del programa. Si se ejecuta el programa tal y como está descrito arriba, obtenemos el siguiente comportamiento: Cuando ya tenemos una primera solución, vamos quitando todas las colas de una en una, subiendo los niveles mirando si todavía queda alguna posible mejor solución: esto es posible porque la **cota es una aproximación**, y no una verdad absoluta.

Si se da el caso que vaciando una cola, encontramos una cota menor que la mejor, ésta se desarrolla y puede acabar en dos vertientes:

1. Se sigue desarrollando hasta llegar a la solución completa y se actualiza como la mejor solución y se continúa.
2. Se descarta en un punto intermedio en el camino.

Tras verlo con muchas pruebas, hemos observado que son exageradamente pocas veces en las que ocurre lo primero, por lo que el algoritmo se suele quedar con la solución conseguida con la aproximación voraz, desarrollando más soluciones parciales sin éxito. El problema es que generalmente para lenguajes neutrales las cotas suelen ser parecidas y, en caso que se desee hacer este procedimiento, se pierde muchísimo tiempo, por ello en el código final se ha implementado también la condición marcada por NOTA1 en el bloque de arriba, por lo que para al llegar a la primera solución.

7.2.4. Cálculo de la cota

Sabiendo cómo funciona el branch and bound, hemos visto que todo depende de la tan mencionada cota, vamos a explicar ahora cómo se calcula y por qué es una aproximación. La cota se calcula mediante el procedimiento de Gilmore-Lawler y se puede dividir en dos términos.

- Primer término: hace referencia al coste del tráfico entre los símbolos (teclas) ya colocados en ubicaciones válidas

- Segundo término: intenta aproximar el coste mínimo que se puede llegar a obtener con las instalaciones aún no emplazadas. Para ello se necesitan dos matrices llamadas C1 y C2, las que se suman, obteniendo C1C2 donde se Hungarian Algorithm para conseguir la asignación óptima y tener una cota más exacta.

- C1: aproxima el coste de emplazar los símbolos (o teclas) que quedan por colocar en ubicaciones libres con los símbolos ya colocados.
- C2: aproxima el coste del tráfico entre todas las instalaciones aún no emplazadas dependiendo de todas las ubicaciones libres.

7.2.4.1. Cálculo del primer término

Este término nos ayuda a ver la cota de la parte que está construída de una solución parcial, en el caso de la solución final, su valor representa la cota final. Para su cálculo, realizamos un cómputo simple recorriendo las *n* instalaciones, y obteniendo su coste respecto al resto de las instalaciones teniendo en cuenta la frecuencia y las distancias entre ellos, si no se ha hecho el cálculo por la parte del otro.

La implementación de esto es la siguiente, otra vez, se ha simplificado respecto a la versión original.

```java
// los parametros nos indican si estan colocados
private int calcularPrimerTermino(boolean[] iesimoSimboloColocado,
                                  int[] jesimaUbicacionDeIesimoSimbolo) {

        int primerTermino = 0;

        for (int simbI = 0; simbI < NUM_OBJETOS; ++simbI) {
            if (iesimoSimboloColocado[simbI]) {          // por cada simbolo no colocado

                for (int simbJ = simbI + 1; simbJ < NUM_OBJETOS; ++simbJ) {
                    if (iesimoSimboloColocado[simbJ]) {  // por cada simbolo colocado por delante
                                                         // acumulamos el valor
                        primerTermino += matFrec[simbI][simbJ] * matDist[ubiI][ubiJ];
                    }
                }
            }
        }

        return primerTermino;
    }
```

Su valor es exacto y la complejidad está acotada superiormente por O(n^2) como se ve por los bucles.

7.2.4.2. Cálculo del segundo término

Para el segundo término, se complican un poco más las cosas porque empezamos a hablar de símbolos que no se han colocado. Como se ha adelantado antes, necesitamos repartirlo en tres pasos, pasos que detallaremos luego.

Igual que el primer término, tenemos que retornar un valor entero, pero para ello nos apoyaremos en dos matrices, que se han de sumar, y otro algoritmo.

Su implementación simplificada es la siguiente:

```java
private int calcularSegundoTermino(int numObjetosColocados, boolean[] iesimoSimboloColocado,
                    boolean[] iesimaUbicacionOcupada, int[] jesimaUbicacionDeIesimoSimbolo) {

// Para evitar sumar C1 y C2, en el calculo de C2 se pasa C1 como parametro y la suma
// se hace implicitamente en calcularSegundoTerminoC2

        int[][] C1 = calcularSegundoTerminoC1(...);
        int[][] C1C2 = calcularSegundoTerminoC2(...);

// Se usa el algoritmo hungaro para encontrar la asignacion optima y con ello obtener el coste
        Hungarian hungarian = new Hungarian(C1C2);
        return hungarian.conseguirCoste();
}
```

7.2.4.2.1. Cálculo de la matriz C1

La matriz C1 se utiliza para aproximar el coste de colocar los símbolos que aún no se han emplazado en ubicaciones libres, teniendo en cuenta los símbolos ya colocados. La implementación de este cálculo es más compleja y se ha simplificado aquí para facilitar la comprensión:

```java
private int[][] calcularSegundoTerminoC1(int numObjetosColocados, boolean[] iesimoSimboloColocado,
                        boolean[] iesimaUbicacionOcupada, int[] jesimaUbicacionDeIesimoSimbolo) {

        int numObjetosRestantes = NUM_OBJETOS - numObjetosColocados;
        int[][] C1 = new int[numObjetosRestantes][numObjetosRestantes];

        int indiceProximoSimboloLibre = 0;          // indice del proximo simbolo no colocado
        for (int i = 0; i < NUM_OBJETOS; ++i) {
            if (!iesimoSimboloColocado[i]) {                // por cada simbolo no colocado

                int indiceProximaUbicacionLibre = 0;    // indice de la proxima ubicacion libre
                for (int j = 0; j < NUM_OBJETOS; ++j) {
                    if (!iesimaUbicacionOcupada[j]) {       // por cada ubicacion libre

   // se calcula el coste como un sumatorio del producto directo entre frecuencias y distancias
                        int sumaCosteIesimoSimboloLibreEnJesimaUbicacionLibre = 0;

                        for (int k = 0; k < NUM_OBJETOS; ++k) {
                            if (iesimoSimboloColocado[k]) { // por cada siombolo colocado
                                  // se va sumando el coste de la varariable sumaCoste
                                  ...
                            }
                        }

                        C1[indiceProximoSimboloLibre][indiceProximaUbicacionLibre] =
                                  sumaCosteIesimoSimboloLibreEnJesimaUbicacionLibre;

                        ++indiceProximaUbicacionLibre;
                    }
                }
```

```
                ++indiceProximoSimboloLibre;
            }
        }

        return C1;
    }
```

Por lo tanto, C1 obtiene valores exactos y el elemento (i, j) de C1 representa el coste de colocar el i-ésimo símbolo (tecla) no emplazado en la j-ésima ubicación libre del teclado, respecto a los símbolos (teclas) ya emplazados. La complejidad de obtener está acotada por O(*n^3*) como se puede observar por los bucles.

7.2.4.2.2. Cálculo de la matriz C2

La matriz C2 se utiliza para aproximar el coste del tráfico entre las todas instalaciones aún no emplazadas, dependiendo de todas las ubicaciones. Como es lógico por su naturaleza, lo que obtendremos de aquí tiene un grado de ambigüedad, ya que queremos ver el coste de emplazar una instalación respecto a las que tampoco están emplazadas; **es por esto que se considera que la cota es una aproximación**.

La implementación es mucho más compleja que las otras a pesar de que la idea es bastante simple: Queremos que el elemento (i, j) de C2 represente el coste de colocar el i-ésimo símbolo (tecla) no emplazado en la j-ésima ubicación vacía del teclado, respecto a los símbolos (teclas) que quedan por colocar.

Para ello, sea T el vector de tráficos desde la i-ésima instalación al resto (de las no emplazadas) y sea D el vector de distancias desde la k-ésima ubicación al resto (de las no ocupadas); la cota inferior del coste de emplazar i en k resulta del producto escalar del T ordenado crecientemente con D ordenado decrecientemente. Es importante organizar bien el cálculo en la implementación porque dependiendo de cuándo se desee ordenar T y D, su complejidad varía. Para minimizarla, en nuestro caso primeramente se calculan y ordenan T y D, y se almacenan en una matriz cada una (cada fila representa un símbolo o ubicación respectivamente).

Estas dos matrices, traficosRestantes y distanciasRestantes, y funcionan exactamente iguales a matrizDistancias y matrizFrecuencias (los atributos de la clase), sólo que éstas son para los símbolos y tráficos restantes y están ordenadas inversamente una a la otra. Esto es porque luego para cada ubicación no emplazada y símbolo no colocado (que son candidatos) hacemos el producto escalar y lo sumamos al valor de C1, obteniendo así C1C2.

Concretamente, la implementación sin entrar en detalles muy específicos, queda así:

Java
```
private int[][] calcularSegundoTerminoC2(int numObjetosColocados, boolean[]
                iesimoSimboloColocado, boolean[] iesimaUbicacionOcupada, int[][] C1) {

        int numObjetosRestantes = NUM_OBJETOS - numObjetosColocados;
        int[][] C1C2 = C1;
```

```
        // la matriz de traficos desde la iesima instalacion al resto de las no emplazadas
        ArrayList<ArrayList<Integer>> traficosRestantes =
                        new ArrayList<ArrayList<Integer>>(numObjetosRestantes - 1);

       // la matriz de distancias desde la jesima ubicacion al resto de las no ucupadas
       ArrayList<ArrayList<Integer>> distanciasRestantes =
                        new ArrayList<ArrayList<Integer>>(numObjetosRestantes - 1);

       // Se rellena y ordena la matriz de traficos restantes, con la ayuda de matrizFrecuencia
        ...

       // Se rellena y ordena la matriz de distancias restantes, con ayuda de matrizDistancias
        ...

        int indiceProximoSimboloLibre = 0;         // indice del proximo simbolo no colocado
        for (int i = 0; i < NUM_OBJETOS; ++i) {
            if (!iesimoSimboloColocado[i]) {            // por cada simbolo no colocado

                int indiceProximaUbicacionLibre = 0;    // indice de la proxima ubicacion libre
                for (int j = 0; j < NUM_OBJETOS; ++j) {
                    if (!iesimaUbicacionOcupada[j]) {   // por cada ubicacion libre

// se calcula el coste como el sumatorio del producto escalar del iesimo vector de traficos y el
// jesimo del de distancias
                        int sumaCosteIesimoSimboloLibreEnJesimaUbicacionLibre = 0;

                        for (int k = 0; k < numObjetosRestantes - 1; ++k) {
                            // se va sumando el coste
                             ...
                        }

// notese que al almacenarlo, se hace '+=' y no '=' porque se suma a los valores obtenidos en C1
                        C1C2[indiceProximoSimboloLibre][indiceProximaUbicacionLibre] +=
                                    sumaCosteIesimoSimboloLibreEnJesimaUbicacionLibre;

                        ++indiceProximaUbicacionLibre;
                    }
                }

                ++indiceProximoSimboloLibre;
            }
        }

        return C1C2;
    }
```

La complejidad es de O(n^3) y viene determinada por el bucle anidado tres veces, ya que es mayor a para ordenar las filas de las matrices, donde se usa un algoritmo de ordenación simple por comparaciones que tiene complejidad O(n log n).

7.2.4.2.3. Mejor asignación de C1C2 mediante Hungarian

La implementación de este algoritmo se ha basado directamente en la fuente que se encuentra en la bibliografía.

El algoritmo implementado en la clase Hungarian resuelve el Problema de Asignación Lineal (LAP) utilizando el algoritmo húngaro. El LAP es un problema de optimización que consiste en asignar de manera óptima un conjunto de tareas a un conjunto de recursos, minimizando el costo total de la asignación. El algoritmo húngaro es especialmente eficaz para resolver este tipo de problemas.

Teniendo como entrada la matriz sobre la que hay que buscar la asignación, el valor de la matriz (i, j) significa el coste de asignar la tarea i, al recurso j, que es lo que tenemos en C1C2.

Todo el proceso se sustenta sobre el siguiente teorema: si un número se añade o sustrae de todos los valores de un columna o fila en una matriz de costes, entonces la asignación óptima para la matriz resultante es también óptima para la matriz original.

Este algoritmo se compone de una fase de preproceso y de una fase iterativa, la implementación no se pone como código porque es muy extensa, pero se puede resumir en lo siguiente:

Paso 1: Reducción de la Matriz

- Se resta el mínimo de cada fila de todos los elementos de esa fila.
- Se resta el mínimo de cada columna de todos los elementos de esa columna.

Paso 2: Marcado de Ceros Independientes

- Se marcan con un "cuadro" los ceros que no tienen otros ceros marcados en la misma fila o columna.

Paso 3: Cubrimiento de Columnas

- Se cubren las columnas que contienen un "cuadro".

Bucle Principal: Paso 4 al Paso 7. Si cada fila y columna de la matriz contiene un cero entonces hemos acabado. Si no, se ejecuta la siguiente fase iterativa hasta que se cumpla esta condición:

Paso 4: Encontrar Z_0 y Marcarlo como "0"*

- Se busca un cero no cubierto en la matriz y se marca como "0*".

Paso 6: Crear Cadena K de "Cuadros" y "0" Alternantes*

- Se crea una cadena K alternando entre "cuadros" y "0*".

Paso 7: Ajuste de Valores

- Se encuentra el valor no cubierto más pequeño en la matriz.
- Se ajustan los valores no cubiertos y se suman a los valores cubiertos.

Verificación y Construcción de la Solución Óptima

- Se verifica si todas las columnas están cubiertas.
- Se construye la asignación óptima a partir de los "cuadros" marcados.

En resumen, el algoritmo realiza iterativamente pasos de reducción de la matriz, marcado de ceros y ajuste de valores hasta que todas las columnas estén cubiertas. Luego, construye la asignación óptima a partir de los "cuadros" marcados en la matriz. El resultado final es una asignación que minimiza el costo total de la tarea. Su complejidad es de O(n^4).

7.2.5. Resultados obtenidos

A nivel experimental, los resultados obtenidos nos han parecido muy buenos teniendo en cuenta que es un problema NP-Completo.

Como se ha descrito en apartados anteriores, principalmente en el de Branch and Bound, la solución que obtenemos no es la mejor entre todas las posibles, demostrar esto para tamaños de alfabeto n grandes es muy costoso. Sin embargo, es una solución prácticamente perfecta que es preferible a lo otro por su gran diferencia en la rapidez con la que acaba.

Esta velocidad, lógicamente, varía dependiendo del lenguaje y del tamaño del alfabeto, pero hemos llegado a una versión en la que generalmente el algoritmo se resuelve en un tiempo inmediato para tamaños de $n < 20$ sin depender del lenguaje.

Para tamaños mayores, hasta $n = 50$, el lenguaje tiene un gran impacto. Si en los textos que componen el lenguaje vemos que hay frecuencias entre símbolos muy definidas, o que son textos cortos o, que no cubren todos los símbolos del lenguaje (si esto ocurre, se puede entender como que realmente la n es menor a la real); la asignación será muy rápida. Sin contar el tercer caso, donde la n es virtualmente menor, si se da uno de los dos casos, nos puede hacer acabar el algoritmo en un tiempo muy pequeño, incluso menor a 5 segundos aún para n=50. En el peor caso, donde el lenguaje es neutral, largo, y cubre todos los símbolos del alfabeto, puede llegar a tardar hasta 1 minuto o algo más para $n = 50$.

Esto es posible principalmente gracias a cómo está implementado el Branch and Bound y el uso que se le da a las colas de prioridad como se ha comentado en su correspondiente apartado.

8. Relación de clases implementadas

Durante la fase de desarrollo de nuestra aplicación, asignamos la codificación de las clases de manera colaborativa. Cada miembro del equipo se encargó de desarrollar clases específicas, abordando así una distribución equitativa de responsabilidades. Esta estrategia permitió una eficiente contribución individual al proyecto.

Esta es la distribución de las clases que ha realizado cada uno de los integrantes del equipo:

- **Guillem Angulo Hidalgo**: DriverGeneral, Tecla, CtrTeclado, VistaAnadirTexto, VistaCambiarNombre, VistaCambiarPosicion, VistaEditarTeclado, VistaModificarLenguaje, VistaVisualizarTeclado

- **Tahir Muhammad Aziz**: Teclado, AlgoritmoDisposicion, Hungarian

- **Joan Martínez Soria**: DriverAlfabeto, DriverGestionDeLenguajes, DriverPalabra, DriverTexto, Lenguaje, Palabra, PalabraEnTexto, Texto, TextoEstandar, TextoFrecuencias, CtrLenguaje, PersistenceAlfabeto, PersistenceLenguaje, PersistenceTeclado, CtrPersistence, VistaCrearLenguaje, VistaCrearTeclado

- *Clases conjuntas*: Alfabeto (Joan y Tahir), CtrDominio (Guillem y Joan), CtrPresentation (Guillem y Joan), VistaPrincipal (Guillem y Joan)

Cabe destacar que en cada una de las clases .java, después de los imports pertinentes, se puede encontrar una muy breve descripción de la misma junto al autor o autores que la han desarrollado

BIBLIOGRAFÍA

- Problema de la asignación cuadrática: https://es.wikipedia.org/wiki/Problema_de_la_asignaci%C3%B3n_cuadr%C3%A1tica
- A branch-and-bound algorithm for the quadratic assignment problem based on the Hungarian method: https://www.sciencedirect.com/science/article/abs/pii/S0377221797000635
- Quadratic Assignment Problems: branch and bound https://nathanbrixius.wordpress.com/2008/05/03/quadratic-assignment-problems-branch-and-bound/
- Hungarian Algorithm https://en.wikipedia.org/wiki/Hungarian_algorithm
- Hungarian Maximum Matching Algorithm https://brilliant.org/wiki/hungarian-matching/
- Sorting while insertion vs Sorting the whole array later https://stackoverflow.com/questions/47944091/sorting-while-insertion-vs-sorting-the-whole-array-later
- HungarianAlgorithm implementation https://github.com/aalmi/HungarianAlgorithm/tree/master
- Interfaz (Java) https://es.wikipedia.org/wiki/Interfaz_(Java)
- Diapositivas de teoría de la asignatura