**Documentación Primera Entrega**

PROP QT 2023-2024

Equipo 33.1

Luis Jesús Valverde Zabaleta

Juan José Torredemer Pueyo

Iván Parreño Benítez

Rafael Ibáñez Rodríguez

**Índice**

[Casos de Uso 3](#_Toc151495390)

[Diagrama 3](#_Toc151495391)

[Descripción 3](#_Toc151495392)

[Diagrama de Clases UML 8](#_Toc151495393)

[Diagrama 8](#_Toc151495394)

[Descripción de las clases 8](#_Toc151495395)

[Teclado 8](#_Toc151495396)

[Lista de Frecuencia de Palabras 9](#_Toc151495397)

[Alfabeto 9](#_Toc151495398)

[Greedy 10](#_Toc151495399)

[Hungarian 11](#_Toc151495400)

[Algoritmo 12](#_Toc151495401)

[QAP/Nodo 12](#_Toc151495402)

[Controlador del Dominio 12](#_Toc151495403)

[Controladores para cada clase 13](#_Toc151495404)

[Relación Clases Por Miembro del Equipo 14](#_Toc151495405)

[Estructuras de Datos y Algoritmos 15](#_Toc151495406)

[Estructuras de Datos 15](#_Toc151495407)

[Alfabeto 15](#_Toc151495408)

[Listas de Frecuencias 15](#_Toc151495409)

[Teclados 16](#_Toc151495410)

[Algoritmo 16](#_Toc151495411)

[QAP 16](#_Toc151495412)

[Nodo 16](#_Toc151495413)

# Casos de Uso

## Diagrama

A diagram of a company

Description automatically generated

## Descripción

**Nombre**: Cerrar aplicación

**Actores**: usuario

**Descripción y diálogo actor-sistema**: Cuando una persona quiere cerrar la aplicación, lo hace saber al sistema mediante un botón de cerrar aplicación y este responde terminando la ejecución de la aplicación.

**Posibles errores**: -

**Nombre**: Gestionar teclados

**Actores**: usuario

**Descripción y diálogo actor-sistema**: El usuario toma la iniciativa de gestionar sus actuales teclados del perfil y lo indica al sistema.

El sistema le da las opciones de crear el teclado, imprimirlo, cambiar su nombre, guardarlo o eliminarlo. Se redirecciona al caso de uso seleccionado.

**Posibles errores**:  El usuario puede cancelar la operación en cualquier momento.

**Nombre**: Crear teclado

**Actores**: usuario

**Descripción y diálogo actor-sistema**: La persona tiene intención de crear un teclado, lo indica al sistema y seguidamente le abre una opción para que pueda hacerlo. El usuario puede asignar un nombre al teclado o de lo contrario usar el predeterminado.

Se redirecciona a escoger lista de palabras, escoger alfabeto, escoger mano y escoger algoritmo.

El sistema crea un teclado con el nombre o no seleccionado y las especificaciones tomadas.

**Posibles errores**: El usuario puede cancelar en cualquier momento y si el nombre seleccionado ya lo contiene otro teclado, aparecerá mensaje de error informando del problema para cambiarlo.

**Nombre**: Escoger mano

**Actores**: usuario

**Descripción y diálogo actor-sistema**: el usuario quiere escoger mano preferente para el teclado y tiene dos opciones, izquierda o derecha. El sistema guarda la elección y cierra la opción.

**Posibles errores**: --

**Nombre**: Escoger alfabeto

**Actores**: usuario

**Descripción y diálogo actor-sistema**: el usuario va a escoger alfabeto y se le abre una opción para que pueda elegir entre el listado, una vez toma la decisión el sistema guarda la elección y lo cierra.

**Posibles errores**: --

**Nombre**: Escoger lista de palabras

**Actores**: usuario

**Descripción y diálogo actor-sistema**: el usuario va a escoger lista de palabras y se le abre una opción para que pueda seleccionar una, el sistema guarda la elección y cierra la ventana.

**Posibles errores**: --

**Nombre**: Escoger forma

**Actores**: usuario

**Descripción y diálogo actor-sistema:** el usuario va a escoger forma, y se abre una ventana con las distintas formas disponibles. Tras la selección de una forma, se guarda la elección y se cierra la ventana.

**Posibles errores**: --

**Nombre**: Escoger algoritmo

**Actores**: usuario

**Descripción y diálogo actor-sistema**: el usuario quiere escoger algoritmo y se le abre una opción para que lo elija, una vez termina se guarda la elección y lo cierra.

Posibles errores: -

**Nombre**: Eliminar teclado

**Actores**: usuario

**Descripción y diálogo actor-sistema**: La persona muestra al sistema que quiere borrar un teclado. Este le permite seleccionar un teclado de todos de los que dispone y la persona toma la decisión.

Una vez seleccionado se abre un mensaje de aviso por si está seguro de la decisión de eliminarlo, advirtiéndole que es borrado permanente. Si afirma que sí se sigue con el proceso y de lo contrario se vuelve abrir la ventana de elección de teclado a eliminar.

**Posibles errores**: Puede cancelar la operación en cualquier momento. Si no existen teclados en el sistema se le avisa con mensaje de advertencia y si el teclado que selecciona está en uso, se vuelve a informar con mensaje de error informando que está en uso y se cancela la operación.

**Nombre**: Guardar teclado

**Actores**: usuario

**Descripción y diálogo actor-sistema**: El usuario indica al sistema que quiere guardar teclado. El sistema guarda el teclado con el nombre indicado o el predeterminado.

**Posibles errores**: Si no hay teclado en uso la funcionalidad será nula.

**Nombre**: Cambiar nombre del teclado

**Actores**: usuario

**Descripción y diálogo actor-sistema**: El usuario quiere cambiar el nombre del teclado y el sistema le abre opción para ello. Selecciona el nuevo nombre para el teclado con el nombre y se guardan los cambios.

Posibles errores:

Puede cancelar en cualquier momento

Si el nuevo nombre es igual que el de otro teclado ya existente, aparecerá mensaje de error informando del problema para cambiarlo.

**Nombre**: Imprimir teclado

**Actores**: usuario

**Descripción y diálogo actor-sistema**: El usuario indica al sistema que quiere imprimir teclado. El sistema muestra en pantalla el teclado solicitado.

**Posibles errores**: Si no hay teclado en uso la funcionalidad será nula.

**Nombre**: Gestionar listas de palabras

**Actores**: usuario

**Descripción y diálogo actor-sistema**: El usuario quiere gestionar las listas de palabras y lo hace saber al sistema. El sistema le muestra un listado de opciones: crear lista de palabras, eliminar lista de palabras, cambiar el nombre de la lista, consultar o modificar palabra.

El usuario escoge una de las opciones y se dirige al caso de uso elegido.

**Posibles errores**:  El usuario puede cancelar en todo momento.

**Nombre**: Crear lista de palabras

**Actores**: usuario

**Descripción y diálogo actor-sistema**:  El usuario indica que quiere crear una lista de palabras, el sistema le da varias opciones, palabra a palabra o a partir de un texto para que elija el usuario

Posibles errores:  -

**Nombre**: Eliminar lista de palabras

**Actores**: usuario

**Descripción y diálogo actor-sistema**: El usuario quiere eliminar una lista de palabras. Elige la lista de palabras que quiere eliminar y lo notifica al sistema.

**Posibles errores**: Puede cancelar la operación en cualquier momento. Si no hay ninguna lista de palabras en el sistema o está en uso se cancela la operación.

**Nombre**: Cambiar nombre lista de palabras

**Actores**: usuario

**Descripción y diálogo actor-sistema**: El usuario indica que quiere cambiar el nombre de la lista de palabras. El sistema le pide el nuevo nombre, el usuario lo indica y se cambia el nombre actual por el indicado

**Posibles errores**: el nombre escogido ya lo tiene una lista existente.

**Nombre**: Consultar palabra

**Actores**: usuario

**Descripción y diálogo actor-sistema**: El usuario indica que quiere consultar palabra. El sistema pide que introduzca la palabra, y le devuelve la palabra junto a la frecuencia de esta en la lista.

**Posibles errores**: la palabra no existe en la lista.

**Nombre:** Modificar palabra

**Actores**: usuario

**Descripción y diálogo actor-sistema**: El usuario quiere modificar una palabra. El sistema pide que introduzca la palabra y la frecuencia deseada de esta en la lista, el usuario escoge y el sistema actualiza la lista.

**Posibles errores**: la palabra no existe en la lista.

**Nombre**: Imprimir alfabetos disponibles

**Actores**: usuario

**Descripción y diálogo actor-sistema**: El usuario quiere imprimir los alfabetos, lo indica y el sistema muestra por pantalla los alfabetos disponibles.

**Posibles errores**: no hay ningún alfabeto.

# Diagrama de Clases UML

## Diagrama

A screenshot of a computer code

Description automatically generated

## Descripción de las clases

### Teclado

La clase teclado es la encargada de definir una instancia de teclado, donde tendremos los siguientes atributos:

* private String nameTeclado;
* private int numeroTeclas;
* private int numFilas;
* int[] filas\_col= new int[numFilas];
* TreeMap<Integer,Character> teclado;
* TreeMap<Integer,Character> teclado\_inverso;
* private int mano;

nameTeclado definirá el nombre del teclado, numeroTeclas definirá el número de teclas del teclado, numFilas definirá el número de filas que tendrá el teclado, y filas\_col será un vector que nos dirá cuántas columnas tendrá cada fila.

Guardaremos la relación letra y tecla en teclado que será un TreeMap<Integer, Character> que nos permitirá tener ordenado de forma creciente según el número de tecla la letra que le corresponde. teclado\_inverso será lo mismo pero invertido según si es para zurdo o diestro que se especificara con la variable mano.

### Lista de Frecuencia de Palabras

Esta clase la usamos para instanciar una lista de frecuencia de palabras para posteriormente poder crear un teclado mediante su algoritmo. La lista de palabras estará compuesta por un mapa, en un apartado posterior comentaremos la razón detrás de nuestra elección. Los atributos son los siguientes:

private *String* name\_list;  
private *Map*<*String*, *Integer*> word\_list;

En nuestro caso el nombre de la lista de palabras lo guardamos en la variable String name\_list y la lista en sí, en un diccionario que tiene un String que es la palabra como clave y un número entero que representa su frecuencia.

En cuanto a las constructoras de la función tenemos varios tipos de maneras de crear la lista:

public Lista\_Palabras (*String name*) {  
 this.name\_list = *name*;  
}

public Lista\_Palabras (*String name*, *String text*) {  
 this.name\_list = *name*;  
 createFromString(*text*);

}

public Lista\_Palabras (*String name*, *Map*<*String*, *Integer*> *freq*) {  
 this.name\_list = *name*;  
 this.word\_list = *freq*;  
}

La primera crea una instancia de la lista de palabras, pero no la inicializa, la segundo crea la instancia y la inicializa en una función que crea la lista a partir de un texto. Por último, la tercera recibe el nombre y un mapa con la lista como parámetro para crear la instancia.

Para acabar en cuanto a los métodos tenemos uno para crear la lista a partir del texto, para imprimir la lista, buscar una palabra, y los getters y setters.

### Alfabeto

La clase alfabeto es al que se encarga de gestionar un vector de caracteres y un nombre, que son sus atributos. En esta primera entrega no hemos considerado la creación de nuevos alfabetos o la modificación de estos.

private char[] caracteres;  
private *String* nomAlfabet;  
private int numCaracters;  
private char primera;

Los métodos de esta clase se encargan de imprimir el alfabeto, y el resto no dejan de ser getters y setters de la clase del modelo.

### Greedy

La clase Greedy se encargará de generar una primera solución, que nos permitirá ahorrar el recorrido de diferentes nodos y empezar desde allí. Nuestro algoritmo Greedy se basa en colocar 4 letras en nuestro teclado. La letra que más peso tiene en la matriz de Flow la colocaremos en la tecla con menor sumatorio de distancias con las otras teclas (habitualmente esta tecla suele ser la del medio). Después de colocar la primera letra, colocaremos las tres que menos influencia tienen en nuestra matriz de Flow, y estas como son las menos utilizadas las colocaremos en las teclas que tienen los mayores costes en el sumatorio de distancias.

La clase Greedy solo tendrá una función pública que se encargara de devolver la solución parcial e inicial, y que de parámetros tendrá la matriz de Distancias, la matriz de Flow y el primer carácter del alfabeto en que se base el teclado, esta es la cabecera de la función solución Parcial:

* public static Map<Integer, Character> Solucion\_Parcial(double[][] distancia, int[][] flow, char primera)

Las estructuras de datos que utilizaremos en esta clase son las siguientes:

* Map<Integer, Character> solucion
* TreeMap<Double, Set<Integer>>  mejores\_teclas
* TreeMap<Integer, Set<Character>> mejores\_letras

El Map de clave Integer y valor Character es un HashMap donde colocaremos la primera solución y que será lo que retorne la clase Greedy al llamarlo. Integer será el numero de la tecla y el Character la letra que se colocará en esa tecla.

El TreeMap de Double y Set<Integer> es un Map **basado en árbol y**  es donde guardaremos las teclas ordenadas de forma descendente. Double representa el sumatorio de las distancias de una tecla determinada con el resto y es la clave del TreeMap, y el valor es el Set<Integer> que representa donde guardamos las teclas con ese mismo sumatorio de distancias ja que diversas teclas pueden tener el mismo sumatorio de distancias.

El TreeMap de Integer y Set<Character> es un Map **basado en árbol** y es donde guardaremos las letras ordenadas en forma ascendente. Integer representara el sumatorio de la letra con su relación con las otras y será la clave del TreeMap, y el valor es el Set<Character> que representa donde guardaremos las letras con el mismo sumatorio de relación con las otras letras, usamos el Set porque puede presentarse el caso de que haya letras con la misma clave.

Inicializo las anteriores tres estructuras y a mejores\_teclas la paso como parámetro junto la matriz de Distancias a la función ***AsignaImportanciaTecla*** que se encargara de hacer el sumatorio de distancias de cada tecla con respecto al resto, y la acumulare en mejores\_teclas, recorreré la matriz de Distancias y por cada fila que representa una tecla, sumare en una variable llamada result todas sus columnas. Después a mejores\_letras hago algo parecido a lo anterior, la paso como parámetro junto a la matriz de Flow y el carácter primera a AsignaImportanciaLetra y se encarga de calcular la influencia de cada letra respecto a las otras, haciendo un sumatorio de sus relaciones, literalmente es lo mismo que la de teclas solo que en vez de recorrer la matriz de Distancias recorro la de Flow.

Luego cuando ya tengo mejores\_teclas y mejores\_letras construidas, llamo a ***ImplementarSolucion*** que se encargara de añadir las teclas con su correspondiente letra en el Map solución.  La letra y tecla más influyentes estarán en el último elemento de sus respectivos TreeMaps y accedemos y conseguimos su valor, y la introducimos en solución. Después mediante iteradores recorro los dos TreeMaps y con un contador inicializado a 0, entro e un bucle while que le meteré como condición que haya next en los iteradores y que contador < 3, después voy relacionando teclas con letras y añado hasta tres parejas de estas.

### Hungarian

La clase Hungarian principalmente solo se basará en calcular el coste de la matriz C3 mediante un conjunto de operaciones que serán descritas a continuación. La uncia función pública será Coste\_Hungarian a la que pasaremos la matriz C3 como parámetro, esta es la cabecera de la función:

* public int Coste\_Hungarian(int[][] matriz)

Tendremos como atributos de clase los siguientes vectores de int[] que nos servirán como indicadores para realizar los pasos para el cálculo del coste final:

* int[] marcaInFila, marcaInCol, filaCubierta, colCubierta, zerosInFila;

Si la matriz que pasamos no es cuadrada, nos encargamos de ampliarla mediante la función ***normalizar*** que le pasaremos matriz, final\_size (será el máximo entre num\_filas y num\_cols), num\_filas y num\_cols.

Para evitar las molestias de copias me genero una matriz\_costes que será una copia de matriz sin modificar y que no compartirán referencia porque hare una Deep Copy.

El siguiente paso será inicializar los atributos de la clase, y poner zerosInFila, marcaInFila y marcaInCol con -1.

Ahora tocara restar el máximo de cada fila a esta misma, y el máximo de cada columna a esta misma también mediante la función ***restar*** y pasándole como parámetro solo la matriz.

Después haremos ***marcar\_zeros*** y ***marcar\_columnas*** que se encargaran respectivamente de marcar los ceros cuando no compartan columna y fila y se indicaran en *marcaInFila y marcaInCol*, y marcar\_columnas que me marcara las columnas que tienen cero asignado y se les dará valor 1, y sino 0.

Después entrare en un bucle while que tendrá de condición la función ***columnasCubiertas*** que devuelve un booleano y que se encargará de iterar por colCubierta y si hay algún elemento con valor 0 retornamos false, y si no hay ningún 0 retornamos true.

Me genero un vector llamado ***mainZero*** que lo inicializaremos con la función ***zero\_sin\_cubrir***, que se encargara de localizar los ceros no cubiertos y meterlos en el vector las coordenadas i, j, y marcarlo en zerosInFila.

Si ***mainZero*** es null calculamos el mínimo que de lo no cubierto y lo restamos a los no cubierto y lo sumamos en las posiciones que tienen columnas y filas marcadas, y volvemos a llamar a zero\_sin\_cubrir.

Después si marcaInFila en la posición mainZero[0] es -1, hacemos el ***marcarRestantes*** que se encargara de modificar macarInCol para posteriormente volver a hacer ***marcarColumnas.***

Si no es -1, cubrimos la fila de mainZero, y desmarcamos la columna de mainZero, y volvemos a calcular el mínimo que de lo no cubierto y lo restamos a los no cubierto y lo sumamos en las posiciones que tienen columnas y filas marcadas.

Cuando acaba el bucle, en marcaInCol tendremos las filas que esta el 0 y la i del bucle nos indicará la columna, lo que nos permitirá ir acumulando en coste los valores respectivos. Después retornamos coste y acaba la función.

### Algoritmo

La clase algoritmo es una clase abstracta que deriva en los dos algoritmos que haremos durante la elaboración del proyecto. En la primera entrega solo hay un algoritmo es por ello que en el diagrama la separación de los tipos aparece como incomplete.

Para realizar esta diferenciación de clases usaremos el patrón strategy ya que cada algoritmo se comporta de una forma diferente.

### QAP/Nodo

El problema por resolver en esta primera entrega es el denominado QAP. La idea de este problema es ubicar n instalaciones, en este caso letras, en n instalaciones que serán las posiciones del teclado.

Para resolver este problema, hemos utilizado el algoritmo denominado Branch&Bound. Cabe destacar que aunque la clase la hemos llamado QAP, el algoritmo que se implementa es el anteriormente mencionado.

La idea general sobre este algoritmo es que se crea un árbol iterativo donde vamos almacenando diferentes soluciones parciales a las que hemos denominado Nodo. Cada nodo tiene un coste al que hemos llamado cota. Además también contiene diferentes TreeSet de Integer y Caracteres donde guardaremos las letras no ubicadas ni las posiciones asignadas. Y por último un TreeMap<Integer,Character> que definiría la estructura del teclado.

Durante la ejecución del árbol iterativo, se crea un Nodo inicial padre proveniente del Greedy, para agilizar el tiempo de computación. Está solución inicial se añade a una priority queue de nodos que será el árbol. A medida que vamos añadiendo letras en posiciones y evaluamos cotas, definimos con cuales queremos o no quedarnos. La idea general es que dado una rama con un nodo padre con valor de cota x, y un primer hijo con valor de cota y, vayamos almacenando los hijos con cota z tal que:( z < x || z  < y). En caso de que todos los hijos tengan peor cota que el padre, nos quedaremos con el hijo con mejor cota y desechamos el resto. Sino, iremos almacenando los mejores hijos. Si hay dos hijos con la misma cota, nos quedaremos con el primero encontrado.

Finalmente, para conseguir la solución, iremos consultando si el nodo que “desencolamos”, el nodo con cota más baja, tiene ya todas las letras asignadas, si es así, hemos encontrado una solución y crearemos el teclado.

### Controlador del Dominio

El controlador del dominio se usa para controlar las instancias del resto de controladores de la capa de dominio. Desde este se llaman todas las funciones asociadas a cada controlador par poder ejecutarlas desde el driver del dominio.

### Controladores para cada clase

Los controladores de las clases del dominio controlan una instancia de la clase a la que controlan de forma que actúan como comunicador entre estas y el controlador del dominio. Los controladores que usamos en nuestro sistema son los atributos del controlador del dominio.

private *CtrlListaPalabras* CDLista;  
private *CtrlAlfabeto* CDAlfabeto;  
private *CtrlTeclado* CDTeclado;  
private *CtrlPersistencia* CDPersistencia;

En este caso el controlador de persistencia es un *stub* que se programará en futuras entregas de este proyecto.

# Relación Clases Por Miembro del Equipo

En este apartado vamos a explicar la repartición de tareas que ha hecho cada miembro. Tenemos que comentar que hemos usado la plataforma de edición de texto IntelliJ Idea y dentro de esta hay una funcionalidad que permite colaborar sobre la misma rama del código a la vez desde diferentes máquinas. Nosotros hemos usado esta opción ya que nuestro conocimiento del git es muy básico y no queríamos perder ningún tipo de información. Debido a esto es posible que los commits estén descompensados y haya clases que parece que las haya hecho un miembro del equipo, pero las ha hecho otro.

|  |  |  |  |
| --- | --- | --- | --- |
| **Iván** | **Rafa** | **Juan José** | **Luis** |
| QAP | ListaPalabras | Alfabeto | Greedy |
| Algoritmo | CtrlDominio | CtrlAlfabeto | Hungarian |
| Nodo | DriverDominio | DriverDominio | Teclado |
| Nodo\_Comparator | CtrlListaPalabras | DriverAlfabeto | CtrlTeclado |
| CtrlPersistencia(Stub) | DriverLista |  |  |

# Estructuras de Datos y Algoritmos

En esta sección del documento vamos a comentar i justificar el porque hemos usado cada estructura de datos en cada clase del modelo. También comentaremos y explicaremos como hemos desarrollado el primer algoritmo para solucionar el problema planteado.

## Estructuras de Datos

### Alfabeto

Un alfabeto siempre es un conjunto de caracteres, para ordenarlo no había muchas maneras, por lo tanto escogimos hacer un vector de chars ya que el coste de recorrerlo no era muy grande, Θ(*n*).

public static final char[] Hebrew = {

'א', 'ב', 'ג', 'ד', 'ה', 'ו', 'ז', 'ח', 'ט', 'י',  
 'כ', 'ל', 'מ', 'נ', 'ס', 'ע', 'פ', 'צ', 'ק', 'ר',  
 'ש', 'ת',

};  
  
public static final char[] Latin = {  
 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j',  
 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't',  
 'u', 'v', 'w', 'x', 'y', 'z',  
};

En este fragmento de código es un ejemplo de como hemos representado el alfabeto hebreo y el alfabeto latín. Como se puede observar lo hemos hecho mediante un vector de caracteres.

### Listas de Frecuencias

Para la clase Lista\_Palabras que representa una lista de frecuencias teníamos varias opciones para representarla, un mapa convencional, o un HashMap entre otras opciones. Resumidamente explicamos cual usaremos y por qué.

* **Sorted Map**

Un sorted map o un mapa ordenado, como indica su nombre es un mapa en el cual las claves están ordenadas por su valor. En nuestro caso las palabras estarían ordenadas según su frecuencia. El Sorted Map también se conoce en Java como Tree Map y la clase se implementa mediante un árbol binario.

Por lo tanto, los costes de búsqueda y recorrido serán de Θ(log(*n*)).

* **HashMap**

El HashMap sería una representación de palabra i su frecuencia por tanto el mapa será de string – entero. Para recorrer el mapa en busca de una palabra el tiempo de acceso medio seria de Θ(1) y en el pero de los caso sería un coste lineal Θ(*n*). Como no necesitamos que nuestro mapa este ordenado ya que para el algoritmo implementado es irrelevante no nos hace falta.

Nos hemos decantado por el HashMap ya que es el que tiene un menor coste de búsqueda y de recorrido de toda la estructura de datos.

### Teclados

La distribución del teclado que crea un algoritmo, en esta primera entrega un TreeMap ya que en el caso del teclado nos interesa mucho que se ordenen las letras de una cierta manera mediante el algoritmo. Por lo tanto el coste asintótico que tiene recorrer un teclado es Θ(log(*n*)).

## Algoritmo

### QAP

Hemos utilizado dos matrices, una de enteros y otra de doubles. En la de enteros almacenamos la matriz de flow y en la de doubles la distancia heurística. Aunque buscamos información respecto a la ley de Fitts y cómo implementarla para el cálculo de esta matriz de distancias, finalmente decidimos dejarlo para futuras entregas, ya que, en esta primera solución, conseguimos resultados correctos.

Utilizamos matrices para almacenar la información de frecuencias y distancias ya que son mucho más eficientes a la hora de acceder a su información cuándo estamos calculando la cota.

Se complementa con el Hungarian y el Greedy, explicado en el apartado de la descripción de las clases del diagrama de clases del modelo.

### Nodo

Cómo se ha mencionado anteriormente, nodo tiene dos TreeSet y un TreeMap almacenados en su clase.

TreeSet<Integer> posiciones\_libres: La idea principal de este Set es ir recorriendo las posiciones libres que quedan por emplazar en orden.

TreeSet<Character> letras\_libres: TreeSet ordenado de las letras que faltan por ubicar.

TreeMap<Integer, Character> posiciones\_ocupadas: A la hora de imprimir el TreeMap, queremos tenerlo ordenado para que escriba las letras en el orden que queremos. Además, durante la ejecución del algoritmo, el tiempo de recorrido del TreeMap es de log n y en todo momento accedemos a valores, en caso de implementar un vector (estructura que se valoró al inicio), pueden existir posiciones que todavía no se ha añadido y por tanto recorrer un vector semi\_vacio.

Nodo, también hace el cálculo de su cota. Primero se calcula el primer término y posteriormente C1 y C2 y se envía su suma C3 = C1+C2 a Hungarian que nos devuelve el coste. Una vez calculado podemos elegir si desechar o no el nodo.