DISSENY D'UN TECLAT

Projecte de Programació

EQUIP 32.4

Segunda Entrega

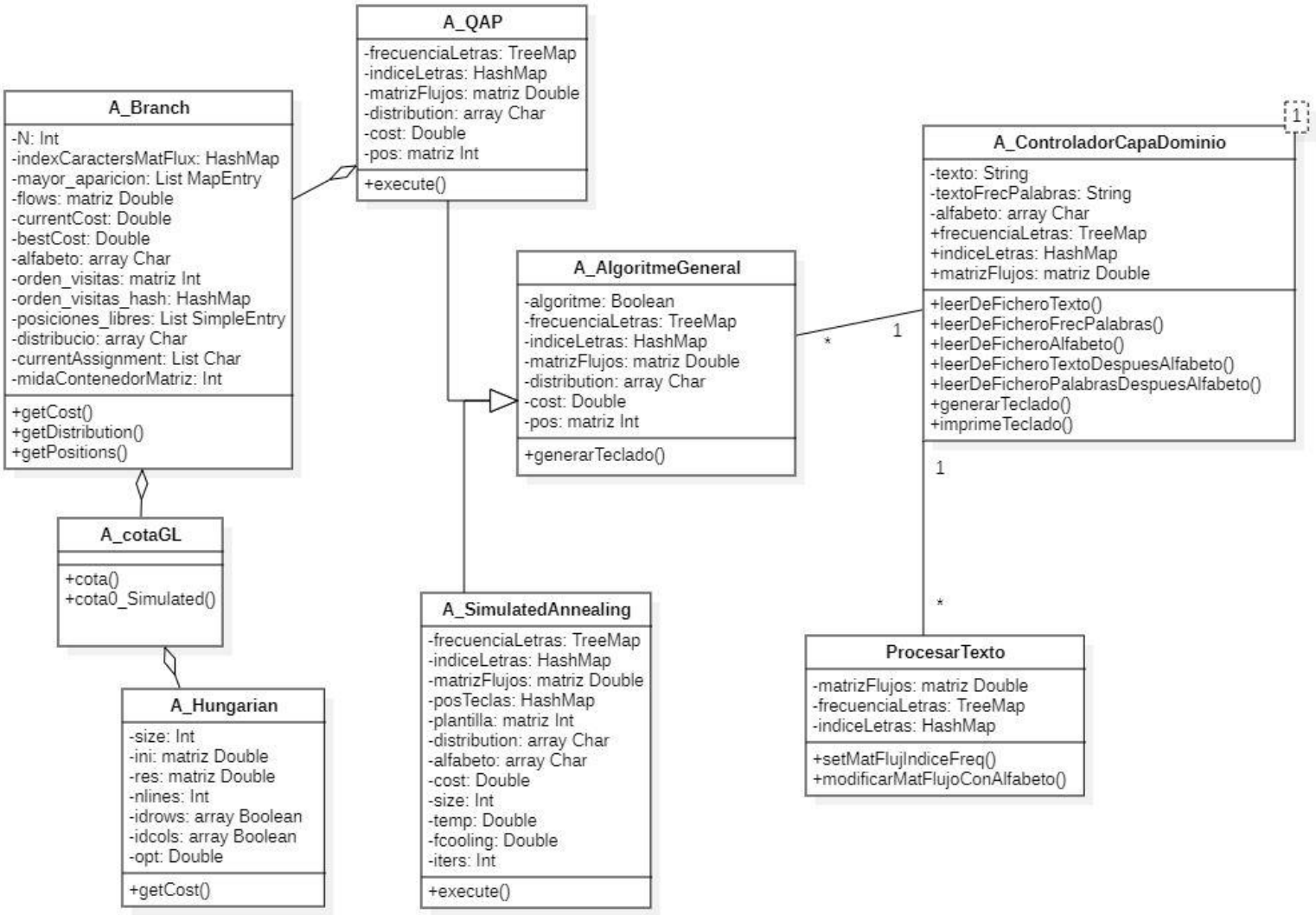
12 de diciembre 2023

Versión Entrega: 1.0

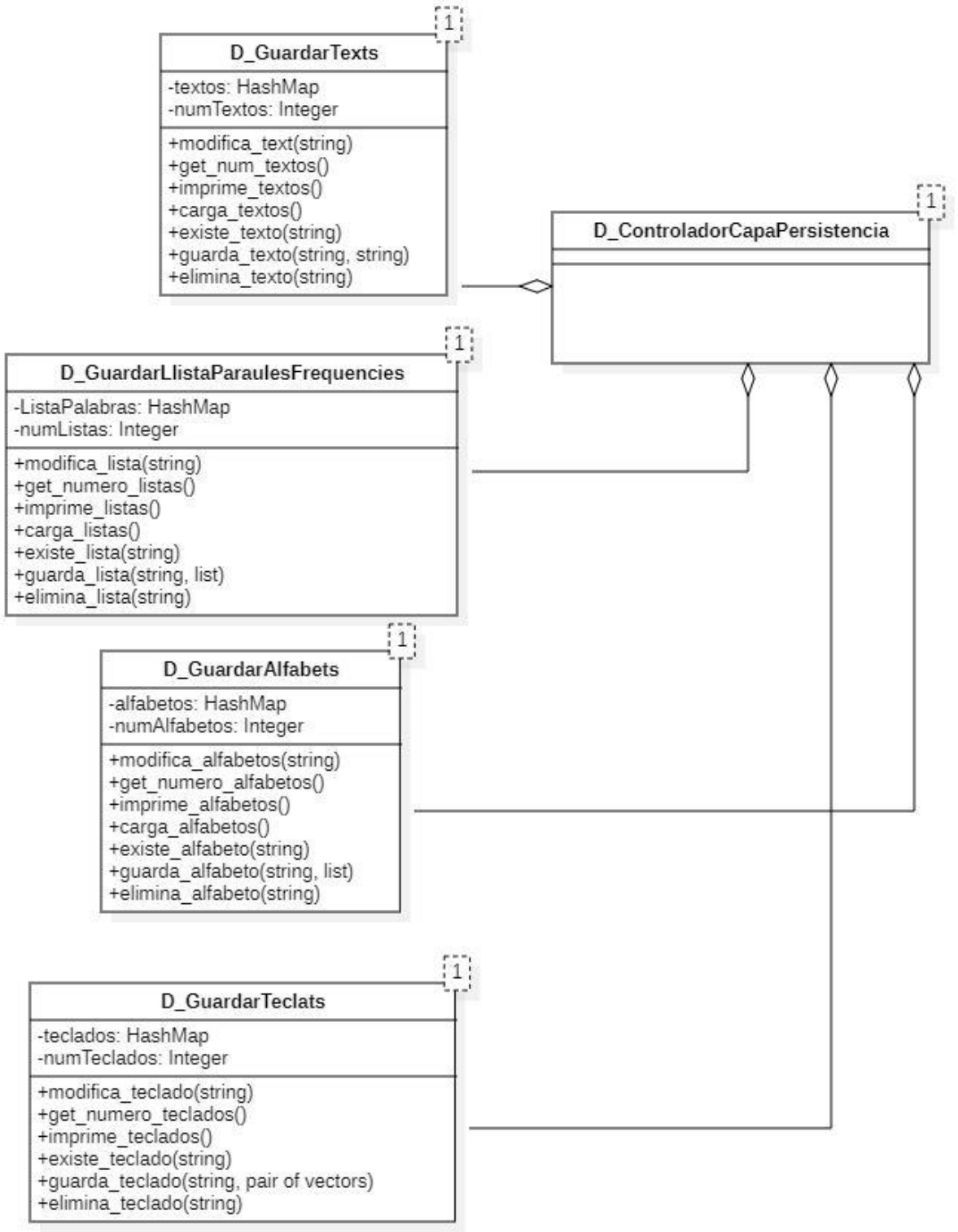
Arnau Claramunt Soler: arnau.claramunt
Joan Marc Coll Barroso: joan.marc.coll
Pablo Franco Carrasco: pablo.franco.c
David García Arévalo: david.garcia.arevalo

ÍNDICE

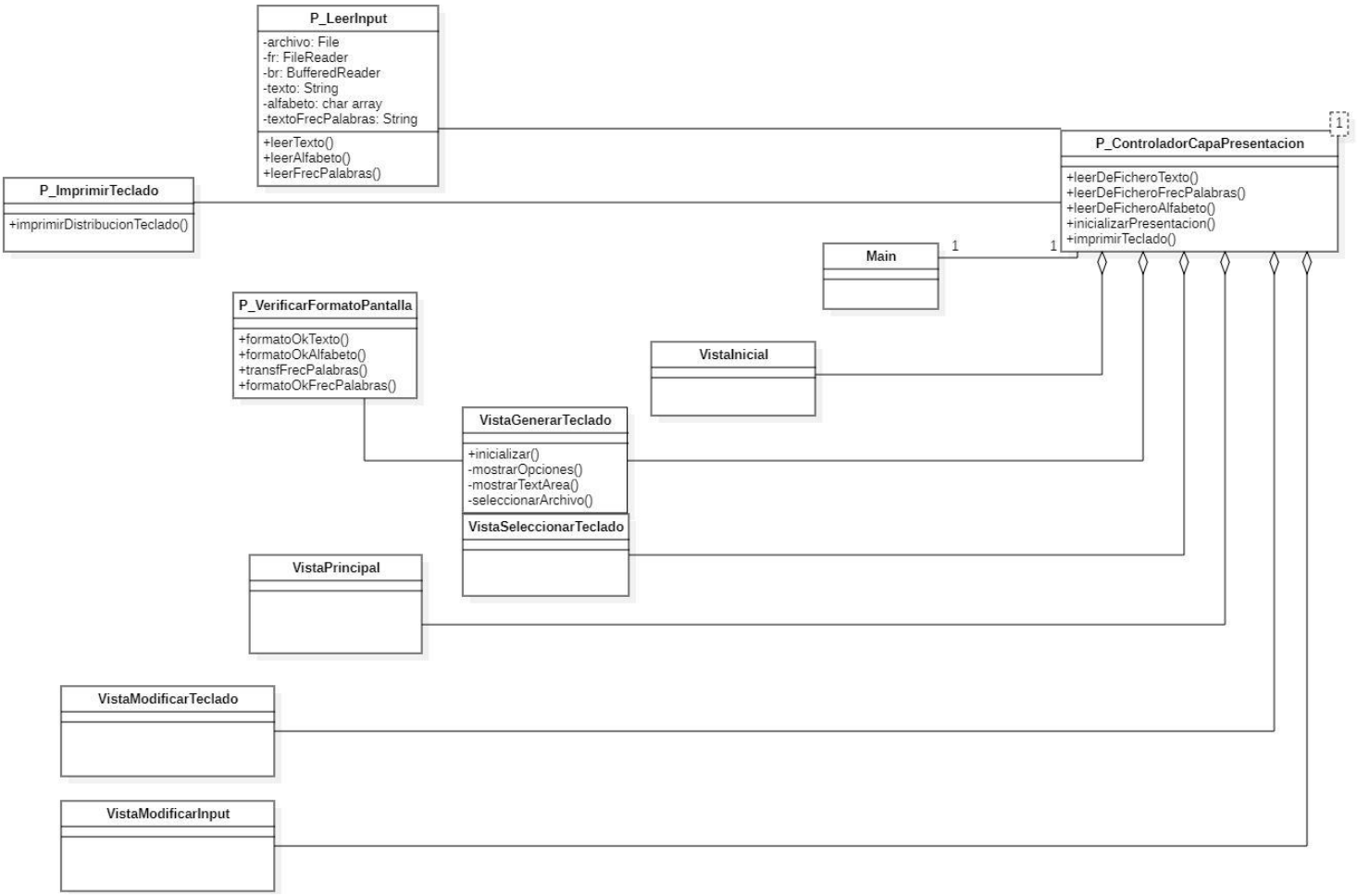
1. Diagrama Capa Dominio



2. Diagrama Capa Persistencia (Gestores de disco)



3. Diagrama Capa Presentacion (Vistas)



4. Descripción Capa Dominio

4.1 Atributos

Clase: A_AlgoritmeGeneral

- **boolean algoritme**: Determina el algoritmo a implementar. Si el tamaño de la entrada del alfabeto es superior a 13, algoritme será False, y ejecutamos un Simulated Annealing implementado con nuestra metaheurística utilizando IA.
  En caso contrario, algoritme será True, e implementamos un QAP con Branch & Bound, teniendo en cuenta la cota de Gilmore Lawler y el algoritmo de Hungarian.
- **TreeMap<Character, Integer> frecuenciaLetras**: Map que nos almacena para cada carácter del alfabeto, las veces que aparece en el texto de entrada.
- **HashMap<Character, Integer> indiceLetras**: Map que nos almacena el índice de las letras para la matriz de flujos.
- **double [][] matrizFlujos**: Matriz de flujos generada
- **char [] distribution**: Vector de chars de la distribución resultante
- **double cost**: Coste de la generación del teclado
- **int [][] pos**: Matriz Nx2 con las posiciones de las teclas colocadas

Clase: A_QAP

- **TreeMap<Character, Integer> frecuenciaLetras**: Map que nos almacena para cada carácter del alfabeto, las veces que aparece en el texto de entrada.
- **HashMap<Character, Integer> indiceLetras**: Map que nos almacena el índice de las letras para la matriz de flujos.
- **double [][] matrizFlujos**: Matriz de flujos generada
- **char [] distribution**: Vector de chars de la distribución resultante
- **double cost**: Coste de la generación del teclado
- **int [][] pos**: Matriz Nx2 con las posiciones de las teclas colocadas

Clase: A_Branch

- **int N**: Número de letras del alfabeto
- **HashMap<Character, Integer> indexCaractersMatFlux**: índice de las letras para la matriz de flujos.
- **List<Map.Entry<Character, Integer>> mayor_aparicion**: Número de veces que aparece cada letra del alfabeto en el texto proporcionado, ordenado de forma decreciente por el valor de aparición.
- **double[][] flows**: matriz de flujos generada
- **double currentCost**: Coste de una solución parcial del algoritmo
- **double bestCost**: Coste de la mejor solución del algoritmo
- **List<Character> alfabeto**: Conjunto total de letras del texto.

- **int[][] orden_visitas**: Posiciones en las que colocaremos teclas ordenadas crecientemente según su distancia al centro del teclado. Este tendrá dimensiones Nx2.
- **HashMap<Character, SimpleEntry<Integer, Integer>> orden_visitas_hash**: Letras colocadas y sus posiciones.
- **List<SimpleEntry<Integer, Integer>> posiciones_libres:** Posiciones del teclado que el algoritmo no ha colocado.
- **char[] distribucio**: Asignación de cada una de las letras en el orden del atributo orden visitas.
- **List<Character> currentAssignment**: Asignación parcial de las posiciones del algoritmo.
- **int midaContenedorMatriz**: Medidas del teclado cuadrado.

Clase: A_hungarian

- **int size**: Tamaño de la matriz. Tendrá como valor el número de filas o columnas
- **double [][] ini**: Matriz con los valores de los elementos iniciales. La necesitaremos para la asignación al final del algoritmo.
- **static double [][] res**: Matriz resultante al aplicar los cálculos. Cada vez que vayamos aplicando cálculos sobre las filas/columnas, se irá sobreescribiendo.
- **static int nlines**: Número de líneas mínimas seleccionadas.
- **static boolean [] idrows**: Filas escogidas por el backtracking.
- **static boolean [] idcols**: Columnas escogidas por el backtracking.
- **double opt**: Coste óptimo de la asignación.

Clase: A_SimulatedAnnealing

- **TreeMap<Character, Integer> frecuenciaLetras**: Map que nos almacena para cada carácter del alfabeto, las veces que aparece en el texto de entrada.
- **HashMap<Character, Integer> indiceLetras**: Map que nos almacena el índice de las letras para la matriz de flujos.
- **double [][] matrizFlujos**: Matriz de flujos generada
- **HashMap<Character, SimpleEntry<Integer, Integer>> posTeclas**: Map que nos almacena dada una letra de nuestro alfabeto, la posición en la que va representada de la matriz teniendo en cuenta el diseño de nuestro teclado.
- **int [][] plantilla**: Matriz de Nx2 que nos da la plantilla sobre qué posición debe ir colocada cada tecla. El vector no se modificará en ningún momento.
- **char [] distribution**: Vector de chars de la distribución resultante.
- **double cost**: Coste de la generación del teclado.
- **char [] alfabeto**: Vector de chars para las diferentes letras/caracteres del alfabeto.
- **int size**: Valor entero para guardar el tamaño del alfabeto.
- **double temp**: Valor de la temperatura del algoritmo.
- **double fcooling**: Valor del factor de enfriamiento del algoritmo.
- **int iters**: Número de iteraciones por cada valor de temperatura.

Clase: ProcesarTexto

- **double[][] matrizFlujos**: aquí guardamos nuestra matriz de flujos
- **TreeMap<Character, Integer> frecuenciaLetras**: aquí guardamos en orden cada carácter que tenemos junto con su frecuencia.
- **HashMap<Character, Integer> indiceLetras**: aquí guardamos el índice de cada char correspondiente a la matriz de flujos.

Clase: A_ControladorCapaDominio

- **P_controladorCapaPresentacion _ctrlCapaPresentacion:** instancia del controlador de la capa presentacion
- **D_ControladorCapaPersistencia _ctrlCapaPersistencia**: instancia del controlador de la capa persistencia.
- **ProcesarTexto pTexto**: instancia de la clase para una vez leído el texto que extraiga las frecuencias entre letras, el alfabeto…
- **A_AlgoritmeGeneral algGeneral:** instancia de la clase Algoritmo General, para poder poner en marcha la generación del teclado.
- **String texto**: para guardar temporalmente el texto leído.
- **String textoFrecPalabras**: para guardar temporalmente el texto con palabras con frecuencias de aparición leído.
- **char[] alfabeto**: para guardar temporalmente el alfabeto leído.
- **TreeMap<Character, Integer> frecuenciaLetras**: las frecuencias una vez se ha procesado, para cuando lo necesite el algoritmo.
- **HashMap<Character, Integer> indiceLetras:** los índices de donde va cada letra en la matriz de flujo una vez se ha procesado, para cuando lo necesite el algoritmo.
- **double[][] matrizFlujos:** con las frecuencias de una letra a otra.

4.2 Métodos

Clase: A_AlgoritmeGeneral:

- **generarTeclado()**: Método que dependiendo del booleano que reciba, llamará al algoritmo de QAP o bien al Simulated Annealing con sus meta heurísticas. Después de ejecutarse, obtendrá los atributos del coste de la generación del teclado, su distribución y las posiciones en las que irán las teclas.
- **getPositions()**: Método que devuelve una matriz Nx2 que representará donde está cada letra colocada una vez se haga la generación del teclado. Le vendrán los datos de la clase A_QAP o bien de A_SimulatedAnnealing.
- **getDistribution()**: Método que devuelve la distribución de las letras del teclado, ordenadas en orden creciente de colocación; es decir, cuanto más a la izquierda, antes se coloca. Le vendrán los datos de la clase A_QAP o bien de A_SimulatedAnnealing.
- **getCost()**: Método que devuelve el coste de la generación del teclado. Le vendrán los datos de la clase A_QAP o bien de A_SimulatedAnnealing.

Clase: A_SimulatedAnnealing:

- **execute()**: Método que llamará a la clase de Simulated Annealing y se irá desencadenando toda la ejecución del algoritmo.
- **swap()**: Operador para intercambiar dos teclas random de nuestra distribución de teclado. Nos servirá para llegar al óptimo global de manera estocástica, ya que Simulated Annealing funciona mejor así que de forma determinista.
- **getPositions()**: Método que devuelve una matriz Nx2 que representará donde está cada letra colocada una vez se haga la generación del teclado.
- **getCostDistribution()**: Meta heurística que nos servirá para minimizar el coste del teclado. Utilizaremos lo que sería la cota0 del Gilmore Lawler, multiplicando para cada letra colocada, la distancia euclidiana por la frecuencia entre ambas sacada de la matriz de flujos. Hay que recordar que la matriz de flujos no es simétrica, por lo que tendremos que tenerlo en cuenta a la hora de calcular esta heurística. Obtendremos finalmente el sumatorio de las distancias entre teclas por su frecuencia
- **getDistribution()**: Método que devuelve la distribución de las letras del teclado, ordenadas en orden creciente de colocación
- **getCost()**: Método que devuelve el coste de la generación del teclado final. Será llamada desde la clase de A_AlgoritmeGeneral para saber cuál ha sido el coste al acabar el algoritmo.
- **spyral()**: Método que nos devolverá una matriz de enteros Nx2 que será la que guardemos en la matriz Nx2 plantilla. Una vez calculadas esas posiciones dado un alfabeto, ya sabremos dónde va colocada cada letra en nuestro diseño del teclado eficiente. Utilizaremos la distribución Expansion Spyral Search (ESS), explicada más adelante.

Clase: A_QAP:

- **execute()**: Método que llamará a la clase de Branching y se irá desencadenando toda la ejecución del algoritmo. Después de ejecutarse, obtendrá los atributos del coste de la generación del teclado, su distribución y las posiciones de las teclas.
- **getPositions()**: Método que devuelve una matriz Nx2 que representará donde está cada letra colocada una vez se haga la generación del teclado. Le vendrán los datos de la clase A_Branch.
- **getDistribution()**: Método que devuelve la distribución de las letras del teclado, ordenadas en orden creciente de colocación; es decir, cuanto más a la izquierda, antes se coloca. Le vendrán los datos de la clase A_Branch.
- **getCost()**: Método que devuelve el coste de la generación del teclado. Le vendrán los datos de la clase A_Branch.

Clase: A_Branch:

- **escogerDistribucion():** determinará según el tamaño del alfabeto una matriz cuadrada cuyos lados sean de longitud impar.
- **calcula_ordre_visites():** recorrerá todas las casillas de la matriz para determinar cuáles son las más cercanas al centro de esta, guardando las N más cercanas en *orden_visitas*.
- **calcula_solucio_inicial():** calculará de forma greedy una solución del teclado para determinar una cota superior del branching.
- **calculaDistribucion():** colocará la primera tecla del teclado y preparará la llamada a branching.
- **branching(level):** función recursiva donde según la cantidad de teclas que se hayan colocado (level) i la cota de la mejor solución encontrada, iterará por las ramas de árbol de espacio de soluciones.
- **getCost():** Devuelve el coste del teclado generado.
- **getDistribution():** Devuelve las teclas colocadas.
- **getPositions():** Devuelve las posiciones correspondientes a las teclas colocadas en *Distribucion*.

Clase: A_cotaGilmoreLawler:

- **cota()**: método general que devuelve un double con la cota calculada a partir de una cota c0, y el valor que devuelve el algoritmo Hungarian, cuando recibe la matriz resultante de la suma de dos matrices C1 y C2.
- **cota0(teclesColocades, indexTeclesMatriuFlux, teclesColocadesIndexPos, flows)**: método que calcula un coste double entre cada par de teclas colocadas. Calculando un sumatorio de coste de distancias por flujos de una letra a otra.
- **cota0_Simulated(teclesColocadesIndexPos, flows, indexTeclesMatriuFlux)**: método adaptado para el algoritmo Simulated, que recibe menos parámetros, y devuelve solo la cota0, ya que todas las teclas están colocadas en el Simulated, el coste de la cota Gilmore-Lawler, solo es de c0.
- **matC1(N, teclesColocades, teclesColocadesIndexPos, teclesPerColocar, indexPosicionsSenseTeclesColocades, flows, indexTeclesMatriuFlux):**

Método que recorre todas las teclas por colocar y por todas las posiciones libres, si se coloca la i-ésima tecla por colocar, como afecta, mirando el coste con todas las teclas colocadas. Es decir las filas de C1 son los caracteres por colocar y las columnas los sitios libres para colocar una tecla, y si se colocara, como afecta a las hechas.

- **precalcularTrafico(numNoColocades,indexTeclesMatriuFlux, teclesPerColocar, flows):** Calcula el tráfico que es el flujo entre dos teclas por colocar
- **precalcularDistancia(numNoColocades, indexPosicionsSenseTeclesColocades):** Calcula las distancias entre teclas no colocadas, para mejorar la eficiencia.
- **matC2(N, m, teclesPerColocar, indexPosicionsSenseTeclesColocades, indexTeclesMatriuFlux):** Principalmente se recorren todas las teclas no colocadas, por todas las casillas por colocar y se calculan los vectores de distancias (D) y tráfico y se hace el producto escalar. Para calcular estos se recorren todas las teclas no colocadas.

Clase: A_hungarian:

- **fasePreproceso()**: Método que nos servirá para hacer los cálculos de la primera fase. Prepararemos la matriz, y se ejecutarán todas las operaciones, por fila y columna, acumulándolas en la matriz resultante.
- **faseIterativa()**: Una vez los cálculos por fila y columna ya han sido realizados correctamente y guardados en la matriz, se asignará el coste óptimo una vez seleccionado el mínimo de líneas por fila y columna.
- **getMaxValue()**: Calcula el máximo valor posible de una matriz. Suponemos que siempre habrá una frecuencia asignada, por lo que existirán números con valor > -1
- **getMinUncovered()**: Calcula el valor mín de los elementos con líneas sin solapar. Nos servirá de utilidad porque se llamará cuando estén trazadas diversas líneas horizontales y verticales.
- **getLinesMin()**: Cálculo de las líneas mínimas necesarias para cubrir todos los ceros de la matriz. Dentro de esta función, inicializaremos los valores pertinentes y a continuación se ejecutará el backtracking para las combinaciones. Esta función está comentada también con un algoritmo propio, pero a falta de una demostración formal, nos hemos decantado por ir adelante con el backtracking clásico para esta primera entrega. El número de líneas lo guardaremos en el atributo static *nlines* y ejecutaremos esta función hasta tener que nlines tiene valor N.
- **getNumZeros()**: Devuelve el número de ceros que quedan por cubrir, dada una combinación hecha con backtracking. Miraremos de ejecutar esta función si el número de líneas encontradas es menor al que teníamos gracias a la ejecución lazy, ya que si no cumple esta primera condición, ya no se ejecutará.
- **getCost()**: Devuelve el coste de la asignación óptima
- **substractValues()**: Restamos los valores por fila y columna de la parte preproceso. Primero obtenemos el valor mínimo de todos los elementos de esa fila/columna y se lo restamos a toda la fila/columna.
- **substractUncoveredValues(double min)**: Restamos el valor a los elementos sin líneas solapadas. El valor del parámetro *min*, será calculado teniendo en cuenta la función mencionada anteriormente *getMinUncovered()*.

- **addSolapatedValues(double min)**: Sumamos el valor a los elementos con líneas solapadas. Lo mismo, el valor del parámetro *min*, será calculado teniendo en cuenta la función mencionada anteriormente *getMinUncovered()*.
- **setMaxValue(double max)**: Asignamos el valor máximo a las posiciones que no haya asignadas un valor. Nos servirá para acabar de formar una matriz NxN para poder hacer los cálculos. Al poner el valor máximo de toda la matriz, no nos afectará en nada porque nunca se escogerá (se busca minimizar el coste de asignación).
- **setCost()**: Asignamos el coste óptimo después de hacer el backtracking con la asignación. Aquí, se hará el segundo backtracking del código que servirá para asignar los ceros de la matriz resultante.
- **initializeValues(double[][] ini)**: Método que sirve para inicializar todos los atributos de la clase, incluyendo la matriz resultante de cálculos que llamaremos *res*, que básicamente será la misma que ini, pero se irá viendo afectada a lo largo del código.
- **initializeLines()**: Se inicializan las líneas de cada fila y columna a false; es decir, pasan a ser no seleccionadas.
- **backtrackingLines(int[] combination, int i, int max)**: Método para calcular las combinaciones de líneas (verticales/horizontales) mínimas que cubran todos los ceros. Al salir de esta función, tendremos las filas y columnas marcadas en *idrows* e *idcols*, respectivamente.
- **backtrackingCost(SimpleEntry<Integer, Integer> [] pos, int i, int act, int size, boolean [] row, boolean [] col, int k)**: Método para asignar los ceros de la matriz para posteriormente sacar el coste óptimo. Como precondición, se asegura que la matriz tiene un cero para cada fila y columna y el número de líneas *nlines* es N. Aseguraremos también que no haya ningún cero escogido de la misma línea (tanto vertical como horizontal) escogido.
- **printMatriu()**: Pinta la matriz resultante después de todos los cálculos para comprobar la correcta ejecución del algoritmo.
- **printDades()**: Pinta los datos del número de líneas escogido, y los vectores de booleanos de líneas (para saber qué filas o columnas han sido las seleccionadas) después de todos los cálculos para comprobar la correcta ejecución del algoritmo.

Clase: ProcesarTexto:

- **setMatFlujIndiceFreq(String texto)**: método que a partir de un texto recibido como parámetro crea la matriz de flujo, junto con el índice de cada letra para esta y también el TreeMap de la frecuencia de cada letra. La función procesa letra a letra el texto para ver cuantas veces va x letra seguida de y así para todas. Por último se divide cada fila entre la aparición de cada letra para tener los valores en su probabilidad. Se ignoran signos de puntuación, solo se tratan letras y espacios.
- **setFrecLetras(String texto)**: método que a partir de un texto recibido como parámetro asigna la frecuencia de cada letra en su atributo TreeMap. Los chars que se tratan son letras de cualquier tipo y el espacio, se ignoran signos de puntuación.
- **modificarMatFlujoConAlfabeto(char[] alfabeto, String texto)**: método el cual a partir del texto y el alfabeto recibidos, modifica la matriz de flujo para añadir en caso de que el alfabeto tenga caracteres que no aparezcan en el texto, añade líneas y columnas a la matriz de flujo con 0.0, ya que no vienen seguidos de otros caracteres. También añade al índiceLletras los caracteres correspondientes junto a

su índice en el HashMap y añade a frecuenciaLetras, el TreeMap, los nuevos caracteres junto a 0 que son las apariciones.

Clase: A_ControladorCapaDominio:

- **A_ControladorCapaDominio():** la constructora donde se inicializan las instancias de los otros controladores y se guardan en los atributos.
- **leerDeFicheroTexto(File fichero)**: lee de un fichero de solo texto que se le indica y procesa su input.
- **leerDeFicheroFrecPalabras(File fichero):** lee de un fichero de solo palabras con frecuencias que se le indica y procesa su input.
- **leerDeFicheroAlfabeto(File fichero)**: lee de un fichero de solo alfabetos que se le indica.
- **leerDeFicheroTextoDespuesAlfabeto(File fichero):** lee de un fichero de texto después que se haya leído de alfabeto y procesa su input.
- **leerDeFicheroPalabrasDespuesAlfabeto(File fichero):** lee de un fichero de palabras con frecuencias después que se haya leído de alfabeto y procesa su input.
- **generarTeclado():** llama al algoritmo general, que este ya se encarga de generar el teclado con el algoritmo QAP o Simulated Annealing
- **imprimeTeclado():** llama al controlador de presentación, que este llamara a la clase P_ImprimirTeclat, para que muestre el teclado hecho. En la primera entrega mostraba por terminal, para la tercera, se mostrará en la interfaz gráfica.

5. Descripción clases Capa Persistencia

Clase: D_ControladorCapaPersistencia
Clase singleton que se encarga de enlazar las clases de la capa de persistencia con las de otras capas, y comunica las distintas categorías de datos que almacena el programa. Contiene diversos getters y setters para acceder y modificar los diferentes datos del programa.

Clase: D_GuardarTexts

Clase singleton que se encarga de gestionar aquellos inputs en forma de texto a la hora de generar un teclado. Esto lo hace a través de un hashmap cuyas claves son los nombres de los teclados a los cuales están asociados. Contiene distintos getters y setters para gestionar la información almacenada.También contienen funciones para cargar y guardar el contenido de los hashmaps en ficheros de texto.

Clase: D_GuardarLlistaParaulesFrequencies

Clase singleton que se encarga de gestionar aquellos inputs en forma de lista con palabras y sus respectivas frecuencias, a la hora de generar un teclado. Esto lo hace a través de un hashmap cuyas claves son los nombres de los teclados a los cuales están asociados. Contiene distintos getters y setters para gestionar la información almacenada.También contienen funciones para cargar y guardar el contenido de los hashmaps en ficheros de texto.

Clase: D_GuardarAlfabets

Clase singleton que se encarga de gestionar aquellos inputs en forma de alfabeto a la hora de generar un teclado. Esto lo hace a través de un hashmap cuyas claves son los nombres de los teclados a los cuales están asociados. Cada alfabeto por sí solo no podrá ser considerado un único input del programa, es por eso que siempre estará asociado a un texto o, a una lista de palabras, esta conexión se forma con el nombre del teclado que usan como llave ambas estructuras, que será el mismo. Contiene distintos getters y setters para gestionar la información almacenada.También contienen funciones para cargar y guardar el contenido de los hashmaps en ficheros de texto.

Clase: D_GuardarTeclats

Clase singleton que se encarga de gestionar los teclados que vaya generando el programa. Esto lo hace a través de un hashmap cuyas claves son los nombres de los teclados en cuestión, y como valores dos vectores que usamos para representar un teclado. Contiene distintos getters y setters para gestionar la información almacenada. También contienen funciones para cargar y guardar el contenido de los hashmaps en ficheros de texto.

6. Descripción clases Capa Presentación

P_ControladorCapaPresentacion:
Se encarga de enlazar las clases de la capa presentación con las de otras capas, también poder comunicarse entre las vistas. El main podría estar situado aquí, pero lo hemos implementado en una clase a parte. También contiene getters de cosas que las clases que necesita. En la tercera entrega, pasará a ser el controlador de más importancia, respecto a la primera entrega que era el de dominio.

Main:
Es la clase general del programa, se encarga de inicializar todo el sistema llamando al Controlador de Presentación, lo pone todo en marcha.

VistaInicial:
Esta vista es la que aparece cuando se ejecuta el programa, se basa en dos botones: uno de Generar Teclado que a partir de aquí se abre la vista de GenerarTeclado, y otro botón Cargar Teclado que a partir de aquí se abre la vista de SeleccionarTeclado. Si ejecutas el programa por primera vez y no tener ningún teclado hecho lo más normal es pulsar a generar Teclado, en caso contrario si no es la primera vez que abres la aplicación, puedes seleccionar y ver un teclado ya creado anteriormente.

VistaGenerarTeclado:
Esta vista se encarga de buscar o introducir el input necesario para generar el teclado, también se decide qué nombre se le quiere poner al teclado. Su pantalla principal está compuesta por unos botones de leer el input según como queramos: por pantalla o por fichero, una cajita para introducir el nombre del teclado (se confirma el nombre ya sea pulsando Enter o pulsando el botón de Confirmar). Y un botón de Generar el teclado que no funciona hasta que los datos estén correctamente introducidos. Los botones de leer no están activados hasta que el nombre del teclado está aceptado. Cuando se pulsa el botón de leer (ambos) se abre una pequeña ventana para elegir que tipo de input va a recibir o escribir, y después si ha pulsado leer por pantalla se abre un cuadro de texto y ahi puedes escribir, y si ha pulsado el de fichero, se abre una ventana para escoger el archivo de tu elección. Si has introducido los datos erróneamente se te comunicará, si lo has hecho correctamente ya podrás generar el teclado.

VistaSeleccionarTeclado:
Esta vista que se abre después de haber seleccionado la opción de Cargar Teclado en la vista inicial, se encarga de, como bien dice el nombre, seleccionar el teclado que queramos de los que tenemos guardados para luego pasar a la vista principal y poder tratarlo como queramos.

VistaPrincipal:
Esta es la vista principal del sistema, una vez tenemos un teclado o bien generado nuevo o bien lo hemos cargado, se abre esta vista donde aparece la distribución del teclado y diversos botones. El botón de Modificar Input que abre la vista de ModifiarInput, el botón de Modificar Teclado que abre la vista de ModificarTeclado, el botón de Cargar Teclado que abre la vista de SeleccionarTeclado, el botón de Generar Nuevo Teclado que nos abre la

vista de GenerarTeclado para poder generar uno nuevo, un botón de Guardar Teclado que guarda el teclado, y uno de Eliminar Teclado que elimina el teclado del sistema.

VistaModificarTeclado:
Esta vista se abre con el botón Modificar Teclado de la vista principal y se encarga de editar el teclado. Por ejemplo, si queremos cambiar la letra x por la letra y. Se indicará que dos teclas se quiere hacer el swap y se podrá guardar el cambio efectuado. Hay que tener en cuenta que si el usuario quiere hacer un cambio, puede que empeore el teclado ya óptimo, no se recalculará, después de hacer el cambio como sería óptimo.

VistaModificarInput:
Esta vista se abre con el botón Modificar Input de la vista principal y se encarga de editar el input que hemos introducido previamente. Hay un botón para guardar los cambios y también para poder volver a la Vista Principal.

P_LeerInput:
Se encarga de leer los datos del fichero ya sea texto, alfabeto o frecuencia de palabras. También verifica que el formato que estamos leyendo sea como queremos y que el archivo que lee no esté vacío.

P_ImprimirTeclado:
Recibe todas las letras en orden que se colocan y por separado su posición, en una matriz que se usan solo dos columnas (realmente lo usamos como pair), y el valor del coste del teclado hecho. En la primera entrega lo mostraba en la terminal y también imprimía espacios en blanco alrededor del teclado ya que es circular, así quedaba con buena forma. Para la tercera entrega se modificará para que se muestre en la Vista Principal una vez el teclado ya esté generado.

P_VerificarFormatoPantalla:
Esta clase se encarga de cuando se lee el texto, el alfabeto o la frecuencia de palabras por pantalla (con la VistaGenerarTeclado), verificar que el formato es el correcto y el que queremos. Se trata en la clase de VistaGenerarTeclado.

7. Descripción Estructuras de Datos

- **List<Character>**
  Lista de caracteres, que se recorre con iteradores. Para guardar secuencias de símbolos.

Ejemplos de uso:
teclesColcocades: todos los caracteres que se han colocado al teclado.

teclesPerColocar: todos los caracteres que faltan por colocar al teclado.

- **double[][]**
  Estructura de datos (ED) 2D, una matriz. Con acceso directo y permite guardar relaciones entre dos variables.

Ejemplos de uso:
flows: es la matriz de frecuencias de ir de un símbolo a otro, es simétrica, cada fila representa un símbolo y lo mismo con las columnas, por ejemplo la frecuencia de la letra ‘e’ a la ‘b’ puede estar en la posición [4][2].

matC1: la matriz del cálculo de una parte de la cota de Gilmore-Lawler, donde se guarda un coste entre las teclas no colocadas respecto a las ya colocadas.

traficPrecalculat: en cada fila hay la frecuencia de ir de una letra no colocada a todas las colocadas.

- **HashMap<Character, Integer>**
  Diccionario que permite guardar una clave y un valor. Los accesos se hacen con la clave y te retorna el valor asociado.

Ejemplos de uso:
indexTeclesMatriuFlux: en este caso en concreto se usa para dado un símbolo, saber su índice de la fila que ocupa en la matriz de flujos. Saber por ejemplo que el ‘g’ esta en la fila 8 de la matriz, y así si se quiere conocer su flujo con otro símbolo ‘u’, se accede otra vez al HashMap, pero esta vez el índice de ‘u’ se interpreta como columna. Y se puede consultar a la matriz de flujo entre ‘g’ y ‘u’ en este orden con: flows[íniceG][índiceU].

- **TreeMap<Character, Integer>**
  Diccionario que permite guardar una clave y un valor ordenados por clave. Los accesos se hacen con la clave y te retorna el valor asociado.

Ejemplos de uso:
frecuenciaLetras: en este caso en concreto se usa para dado un carácter, saber su frecuencia de aparición.

- **SimpleEntry<Integer, Integer> []**
  Estructura de datos que permite guardar una pareja, en este caso de dos enteros.

Ejemplos de uso:
pos: es el vector que contiene todas las asignaciones de los ceros escogidos (para el algoritmo de Hungarian). Al tener sus posiciones {i, j} almacenadas, podremos sacar el coste óptimo sumando de la matriz inicial los elementos situados en dichas posiciones.

- **List<Map.Entry<Character, Integer>>**

  Lista que permite guardar una clave y un valor asociado, respetando el orden de inserción. Los accesos se realizan a la posición que ocupa cada elemento en la lista igual que un vector, siendo 0 la posición del primer elemento insertado.

Ejemplos de uso:
mayor_aparicion: en este caso se usa para tener ordenadas decrecientemente las letras de un alfabeto según su frecuencia de aparición.

- **HashMap<Character, SimpleEntry<Integer, Integer>>**
  Diccionario que permite guardar una clave y un valor, siendo este valor una pareja de dos elementos. Los accesos se hacen con la clave y te retorna el valor asociado.

Ejemplos de uso:
orden_visitas_hash: en este caso en concreto se usa para dado un carácter, conocer su posición en la matriz del teclado.
posTeclas: en este caso en concreto se usa para dado un carácter, conocer su posición en la matriz del teclado (pero implementado en el Simulated Annealing).

- **HashMap<String, String>**
  Diccionario que permite guardar una clave en forma de string, asociada a su valor que es otro string.

Ejemplos de uso:
textos: permite almacenar en la capa de datos aquellos input en forma de texto, cuya clave es el nombre del teclado que generan.

- **HashMap< String , List < SimpleEntry<String, Integer> > >**
  Diccionario que permite guardar una clave en forma de string, asociada a una lista, donde cada elemento es un pareja formada por un string y un entero.

Ejemplos de uso:
lista_palabras_frec: usada en la capa de datos para almacenar aquellos input en forma de lista con cada palabra asociada a una frecuencia de aparición de esta, donde la clave de la estructura será el nombre del teclado que generan.

- **HashMap< String , List <Character >>**
  Diccionario que permite guardar una clave en forma de string y un valor, siendo este una lista de carácteres.

Ejemplos de uso:
alfabetos: usada en la capa de datos para almacenar aquellos aquellos alfabetos dados como input del programa, donde la clave de la estructura será el nombre del teclado que generan.

- **HashMap< String , SimpleEntry <char[] , SimpleEntry<Integer, Integer>[]>>**
  Diccionario que permite guardar una clave en forma de string y un valor, siendo este valor una pareja de dos elementos, uno es un vector de carácteres, y el otro un vector de parejas de enteros.

Ejemplos de uso:
teclados: usado en la capa de datos para almacenar los teclados generados por el programa, donde la i-ésima posición del vector de pares de enteros, representa la posición del teclado de la tecla que encontramos en la i-ésima posición del vector de carácteres. La clave de estos teclados será el propio nombre que tenga asociado el teclado.

8. Descripción Algoritmos

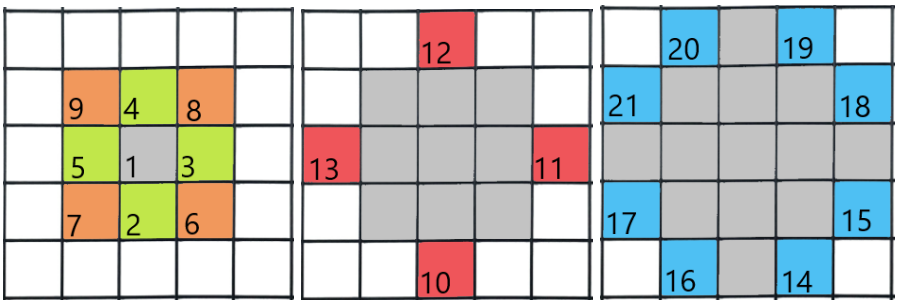
8.1 Branching & Bounding

El algoritmo Branch and Bound es una técnica de búsqueda exhaustiva utilizada para resolver problemas de optimización combinatoria. Su principal objetivo es reducir el espacio de búsqueda explorando solo aquellas soluciones prometedoras y descartando aquellas que no tienen posibilidad de ser mejores que la mejor solución encontrada hasta el momento.

El proceso de Branch and Bound implica dividir el espacio de búsqueda en subconjuntos más pequeños, explorar estos subconjuntos y podar aquellos que no puedan contener una solución óptima. Aquí está el proceso detallado de cómo hemos implementado este algoritmo :

1. Inicialización:
   - Hemos determinado las dimensiones del teclado de nuestra solución, y decidido un orden de posiciones en el cual se colocarán las teclas de la solución en función de su distancia al centro del teclado.

Para la distribución donde se pondrán las teclas, hemos decidido que será circular y se construye con un Algoritmo **BFS modificado:** bautizado por nosotros como: ESS (Expansion Spyral Search), donde se parte del centro de la matriz, y se va expandiendo usando una algoritmo de BFS corriente, con la peculiaridad de que cada casilla se añade a una cola de prioridad ordenada por la distancia al centro; consiguiendo así tratar primero las celdas vecinas más cercanas de la cuadrícula (Grid), dandole un efecto de espiral. Así se consigue que todo esté cerca respecto al centro, al tener un solo dedo con el teclado interesa que todo se construya a partir de la tecla del medio. (A la hora de imprimir el teclado como se hacía en la primera entrega, realmente, está dentro de una matriz que hace de contenedor exterior, para que también se impriman los espacios en blanco de los alrededores y el teclado circular colocado dentro).



Orden que se sigue, al inicio se miran adyacentes y diagonales respecto el centro, luego ya solo adyacentes de cada celda por orden. (El orden para los adyacentes es: abajo, derecha, arriba, izquierda. Pero este orden podría ser cualquier otro) (En la representación de la izquierda, la matriz o caja contenedora sería de 3x3 donde caben 9 teclas y es simétrico, y cuando se añade otra tecla, ya hay que ampliar todos los border una unidad, lo que significa que hay una fila más por arriba y abajo y lo mismo para las columnas izquierda y derecha, haciendo que aumente a un 5x5, con un máximo

de 25 teclas, observar que siempre son números impares, por los tanto el centro será exacto (numeroFilas/2, numeroColumnas/2)). No se usa un BFS normal, porque desde la posición 2 exploraria la 10, y esta se aleja mucho, antes habría que haber tratado las diagonales.

También al usar las distancias euclidianas al cuadrado se puede ver las proximidades de las celdas respecto el orígen, haciendo que el teclado circular sea óptimo para minimizar las distancias. También esto indica claramente que es mucho mejor que usar la Manhattan distance, ya que esta no contempla que por ejemplo la diagonal, North-East del centro está más cercana que las dos celdas adyacentes a la derecha del centro.

	5	4	5	
5	2	1	2	5
4	1	0	1	4
5	2	1	2	5
	5	4	5	

Distancias al cuadrado del origen a las otras celdas

(para calcular los valores se puede suponer que el origen está en la posición (0,0) y la celda de la derecha del centro está a la (0, 1) por ejemplo).

- De una forma **greedy** hemos colocado en orden aquellas teclas con mayor frecuencia de aparición en el texto, en las posiciones más cercanas al centro del teclado. Calculando al final el valor de la cota de Gilmore-Lawler de esta solución, para obtener una cota superior para las soluciones de nuestro espacio de búsquedas.
- Una vez calculada la cota de la solución inicial, empezamos el algoritmo de branching colocando de forma greedy la tecla con mayor frecuencia de aparición en el centro del teclado.

2. Búsqueda:
- Selecciona el subconjunto de las letras del alfabeto que aún no se han colocado en la solución parcial.
- Se generan subproblemas probando de colocar cada una de las letras no emplazadas en la próxima posición del teclado.
- Exploramos cada uno de los subproblemas de manera recursiva y siguiendo un criterio Lazy, es decir, que retrasamos el cálculo de las cotas de los subproblemas hasta que sea estrictamente necesario.

3. Cálculo de límites:
- Para cada subproblema generado, se calcula su cota de Gilmore-Lawler que proporciona un límite superior en la calidad de la solución.

- Si la cota superior de un subproblema es peor que la mejor solución encontrada hasta el momento, se descarta este subproblema y no se explora más. En caso contrario, pasaremos a explorar la rama de este subproblema.

4. Actualización de la mejor solución:
   - Durante el proceso, se actualiza continuamente la mejor solución encontrada cuando llegamos a una solución completa, es decir, que tenga emplazadas todas las teclas.

5. Terminación:
   - El algoritmo termina cuando se ha explorado todo el espacio de búsqueda relevante.

8.1.1 Mejoras respecto a la primera entrega

Ya que hay casos en los que como input se nos proporciona además de una lista de palabras o un texto, un alfabeto; y en estos casos es posible que el alfabeto contenga letras que no encontramos en la otra parte del input (letras sin frecuencia de aparición), lo que haremos será excluirlas del algoritmo, para más tarde añadirlas lo más alejadas posible del centro del teclado.

8.2 Cota de Gilmore-Lawler

La cota de Gilmore-Lawler, su objetivo en este proyecto es obtener una cota de cómo de bueno es colocar una tecla en un sitio libre del teclado. Obteniendo un valor double, y para hacerlo tiene en cuenta las teclas ya colocadas entre ellas, al colocar una nueva tecla como afecta a las ya colocadas, y las no colocadas entre sí.
Esto se divide en tres partes la cota, como se acaba de mencionar en orden: una cota 0 (c0), una matriz C1 y una matriz C2.

Principalmente los cálculos se basan en la distancia por frecuencia de una letra a otra.

En c0 se mira por cada par de teclas colocadas un acumulado de la distancia entre ellas multiplicada por el flujo. Teniendo un coste O(m^2), donde m es el número de teclas colocadas.

Para calcular la segunda parte de la cota se calcula una matriz C1 donde para hacer más eficiente se precalculan las distancias, ya que se repiten en cada columna de C1. Y principalmente se mira por toda tecla no colocada, por todas las posiciones libres donde puede ir, y si se colocara allí, calcular el producto distancia por flujo, con cada tecla ya colocada. Y este es el valor que va en cada casilla de C1, es el coste de colocar la i-ésima tecla, en la k-ésima posición respecto todas las ya colocadas. Con un coste $O((N-m)^2 \cdot m)$, donde N es todas las teclas, por tanto N-m son las no colocadas

Para calcular la tercera parte de la cota se calcula una matriz C2, donde usa tráfico y distancias que son posibles de precalcular. El vector de tráfico (T) se construye con el flujo entre dos teclas no colocadas, en el código se explican los detalles.

Y principalmente se recorren todas las teclas no colocadas, por todas las casillas por colocar y se calculan los vectores de distancias (D) y tráfico y se hace el producto escalar. Para calcular estos se recorren todas las teclas no colocadas, por tanto el coste de la parte de C2 es $O((N-m)^3)$.

Finalmente se suma C1 y C2, para buscar una asignación óptima y ese resultado de la cota de la asignación óptima, sumado con c0, da el valor de la cota de Gilmore-Lawler.

8.3 Algoritmo de Hungarian

El algoritmo de Hungarian se divide en dos fases; la fase preproceso y la fase iterativa. Este algoritmo es bastante utilizado en problemas de asignación. En relación a la generación del teclado, nosotros lo utilizaremos para sacar el coste óptimo, dada una matriz de costes donde intervienen las cotas anteriormente explicadas.

La primera de las fases a las que se somete nuestra matriz es la preproceso. Aquí digamos que es donde se prepara la matriz para hacer todos los cálculos posteriormente. Lo primero a realizar es asignar el valor máximo de la matriz a aquellas posiciones que no cumplen la condición de ser matrices NxN, con el objetivo de buscar esa distribución.

Después, para cada fila obtenemos el coste mínimo y se lo restamos a cada elemento de esa fila. Observamos que en las posiciones donde permanece ese elemento mínimo ahora habrá un 0, la cual cosa nos vendrá bien para más adelante. Seguidamente, hacemos el mismo proceso para las columnas.

Llegados a este punto, la fase preproceso ha finalizado. Nos adentramos en la fase iterativa. Tenemos que ver que haya un cero para cada fila y columna. Por contra, deberemos calcular las líneas mínimas necesarias para cubrir todos los ceros, ya sean filas o columnas. Este procedimiento deberemos repetirlo hasta conseguir que las líneas mínimas sean igual a N (tamaño de fila o columna de la matriz).

Aquí, realizaremos un backtracking clásico para la asignación de filas y columnas. Pero, dentro de la clase, si nos detenemos a ver el código, podremos observar una distinta implementación con un algoritmo greedy pensado por un miembro del grupo, diferente al de backtracking. Cabe destacar que finalmente no hemos podido utilizarlo debido a que tendríamos que haber realizado una demostración formal, pero el algoritmo funcionaba con todas las matrices que le pasabamos. El motivo de haber dejado finalmente el algoritmo de backtracking es porque con el greedy no nos aseguraría formalmente que el número de líneas seleccionado fuera el mínimo al escoger una línea con la misma prioridad que otra (siendo una fila y otra columna). Por tanto, decidimos pagar el coste temporal del backtracking, pero nos aseguramos tener la mejor solución.

Una vez sacamos las líneas mínimas necesarias para asignar todos los ceros y vemos que no es igual al número N, tenemos que seguir calculando una serie de cosas. La primera es obtener el valor del elemento mínimo de la matriz resultante sin que esté cubierto por ninguna línea (séase horizontal o vertical). Después, restamos a todos esos elementos no cubiertos el elemento mínimo, pero solo una vez. A continuación, sumamos el valor del elemento con el valor mínimo de los no cubiertos y lo sumamos a las líneas que estén

solapadas; es decir, que para un elemento, haya marcada una línea horizontal y a la vez vertical. Hacemos nuevamente el conteo de líneas mínimas.

Si el resultado sale que las líneas mínimas necesarias para cubrir todos los ceros tiene valor N, hemos acabado gran parte de los cálculos. Por contra, deberemos repetir el procedimiento anterior.

En caso de acabar, asignaremos los ceros que hayan en la matriz resultante con otro backtracking para la asignación. Nos guardaremos las posiciones escogidas y el coste óptimo, que será mínimo, lo calcularemos ayudándonos de la matriz inicial, porque la matriz no mueve a los elementos de sitio, simplemente hace cálculos con ellos dependiendo del resto de elementos. Finalmente, el coste resultante podemos asegurar que es el óptimo.

8.4 Simulated Annealing

Con el objetivo de obtener una solución óptima al problema de la creación del teclado en un tiempo razonable, nos vemos favorecidos al ejecutar el algoritmo de Simulated Annealing. Por tanto, ejecutaremos dicho algoritmo a partir de un array de alfabeto de tamaño 13.

Partiendo del nombre del algoritmo, el término “annealing” hace referencia a “recocido”. Es un algoritmo de optimización inspirado en la metalurgia, específicamente en el proceso de recocido. La idea principal es simular el recocido de un material, donde se calienta y luego se enfría gradualmente. En lugar de buscar directamente la mejor solución, explora diferentes opciones y, a veces, acepta soluciones no tan buenas al principio. Por tanto, el algoritmo busca encontrar el mínimo global explorando el espacio de soluciones, aceptando también soluciones subóptimas, para no estancarse en mínimos locales.

Inicialmente, obtendremos una distribución de teclado. Nos dará igual cuál sea, pero el único requisito será tener colocadas todas las teclas. Esa será nuestra solución inicial.

Para calcular el coste usaremos la meta heurística de la cota 0 de Gilmore Lawler, que básicamente multiplicará para cada letra del teclado, la distancia euclidiana por el flujo que haya entre cada par de letras.

Sea c1 el coste del teclado de la solución inicial. Aplicando el operador de swap, que lo que hará será swapear dos teclas random, obtendremos otra distribución, con coste c2. Compararemos costes, sabiendo que será mejor teclado aquel que tenga menor coste. ¿Es menor c2 que c1? En caso afirmativo nos quedamos con la nueva distribución. Por contra, tenemos que ver si también debemos explorar la solución, aún empeorando el coste, por si a la larga encontramos soluciones más óptimas en el espacio de soluciones.

Iremos bajando la temperatura para que se vaya enfriando y acercando cada vez más al mínimo global. Para cada valor de temperatura haremos un número de iteraciones constante, donde en cada una de ellas aplicaremos nuestro operador de swap y mejorando el coste del teclado; es decir, llegando a la distribución óptima.

Una vez acabadas estas iteraciones, multiplicaremos la temperatura por un factor de enfriamiento para ir disminuyendo la temperatura.

Estos valores pueden ser muy variados, y serán prefijados de manera experimental. Para tener una orientación de la magnitud de estos, para la temperatura podemos empezar en 100, ejecutar 10.000 iteraciones por temperatura y que nuestro factor de enfriamiento sea un valor entre 0.90-0.99, por ejemplo. Nótese que se pueden hacer muchas combinaciones con estos valores, ya que por ejemplo, cuanto más alto sea este factor de enfriamiento, más lento enfriará. El algoritmo acabará cuando el valor de la temperatura sea inferior a 1.