DISSENY D'UN TECLAT

Projecte de Programació

EQUIP 32.4

Primera Entrega

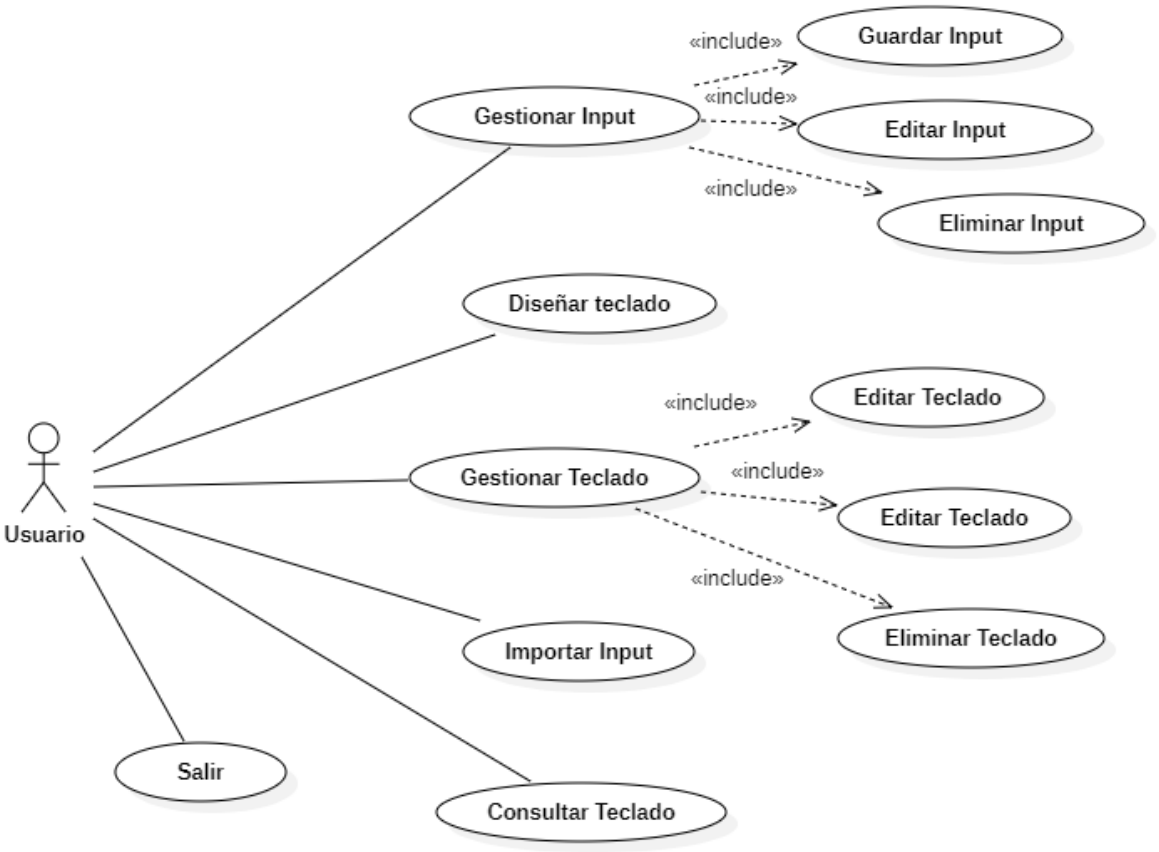
21 de noviembre 2023

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 Casos de uso



1.1 Descripción Casos de uso

Nombre: Importar Input

Actor: Usuario

Comportamiento:
-El usuario proporciona un texto o bien un alfabeto, una lista de frecuencia de aparición de cada letra y una matriz de flujos entre letras.
-El sistema valida los valores y, en caso de ser un texto, lo descompone para extraer un alfabeto, una lista de frecuencia de aparición de cada letra y una matriz de flujos.

Errores posibles y caminos alternativos:
-Si hay alguna incongruencia entre las letras proporcionadas por el alfabeto, la lista de frecuencias y la matriz de flujos, se solicitará al usuario un input válido.

Nombre: Gestionar Input

Actor: Usuario

Comportamiento:
(Se implementará en próximas entregas)
-El usuario podrá guardar, editar y eliminar, según su interés, los inputs que ha proporcionado durante la ejecución del programa.

Nombre: Diseñar teclado

Actor: Usuario

Comportamiento:
-El usuario selecciona para qué input proporcionado desea diseñar un teclado.
-El usuario introduce qué algoritmo se utilizará para este diseño.

Errores posibles y caminos alternativos:
-Si el input proporcionado no existe, el sistema preguntará de nuevo al usuario.
-Si el algoritmo seleccionado no existe, el sistema volverá a preguntar al usuario.

Nombre: Gestionar Input

Actor: Usuario

Comportamiento:
(Se implementará en próximas entregas)
-El usuario podrá guardar, editar y eliminar, según su interés, todos los teclados diseñados durante la ejecución del programa.

Nombre: Consultar teclado

Actor: Usuario

Comportamiento:
-El usuario seleccionará uno de los teclados guardados.
-El sistema validará que el teclado exista y lo mostrará por pantalla.

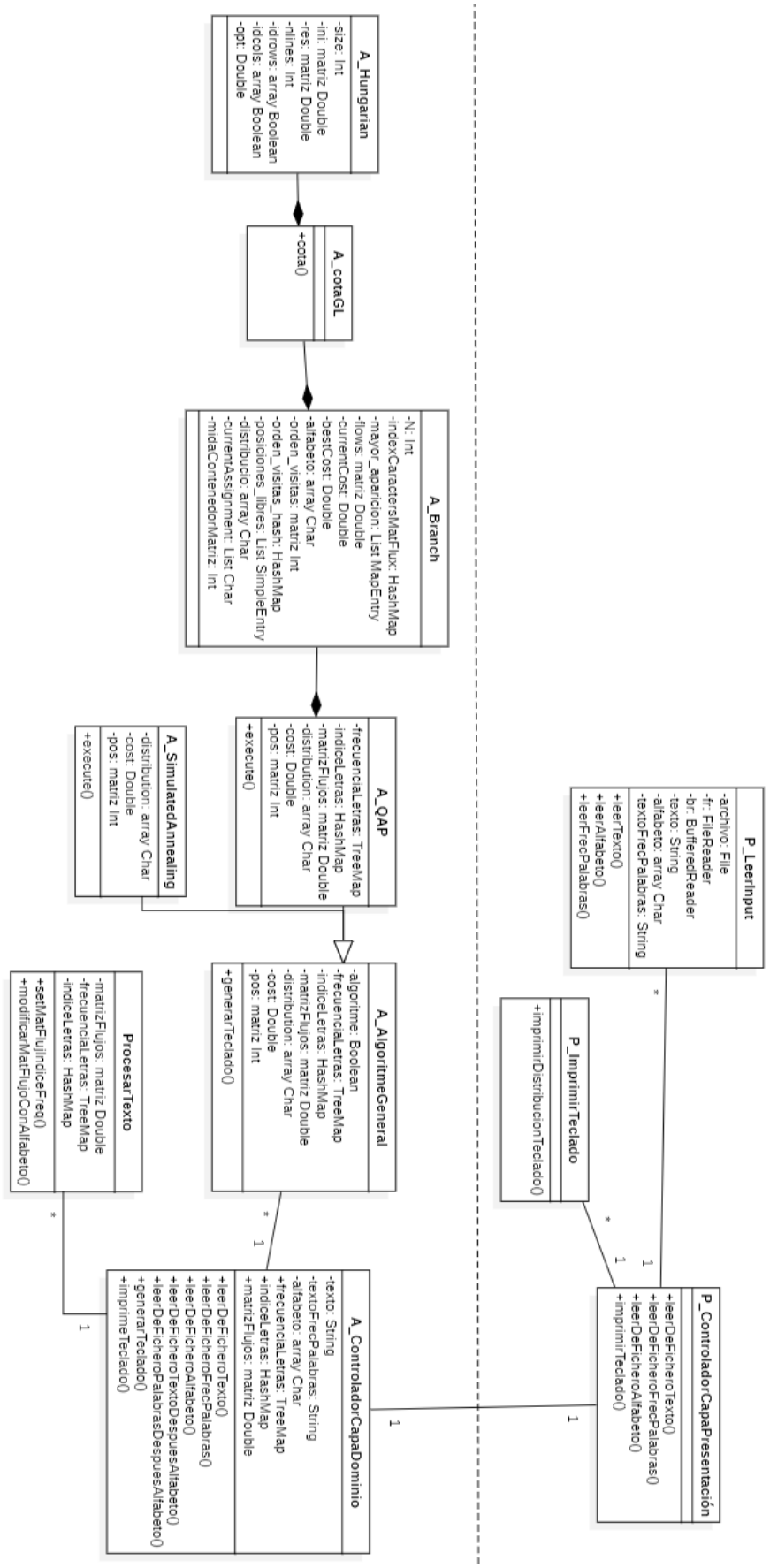
Errores posibles y caminos alternativos:
-Si el teclado seleccionado no existe, el sistema volverá a preguntar al usuario por un teclado para mostrar.

Nombre: Salida

Actor: Usuario

Comportamiento:
-El usuario indicará al sistema que desea abandonar la ejecución del programa.
Después de confirmación, el sistema se cerrará.

2. Diagrama de Clases, versión diseño



2.1 Descripción atributos

Clase: A_AlgoritmeGeneral

- **boolean algoritme**: determina el algoritmo a implementar.
  (True -> QAP || Fasle -> Simulated Annealing)
- **TreeMap<Character, Integer> frecuenciaLetras**: para tener en consideración la frecuencia de aparición de las letras.
- **HashMap<Character, Integer> indiceLetras**: 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**: para tener en consideración la frecuencia de aparición de las letras.
- **HashMap<Character, Integer>**: 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**: Numero 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 (filas, columnas)
- **double [][] ini**: matriz inicial de los elementos (la necesitaremos para la asignación al final del algoritmo)
- **static double [][] res**: matriz resultante al aplicar los cálculos (se irá sobreescribiendo)
- **static int nlines**: número de líneas mínimo 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

- **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: P_LeerInput

- **File archivo:** se usa para el nombre (ruta) del archivo que queremos tratar.
- **FileReader fr**: para lector el archivo.
- **BufferedReader br**: para el buffer del lector de archivo.
- **String texto:** guardaremos aquí el texto que leemos.
- **char[] alfabeto**: guardaremos el alfabeto leído aqui.
- **String textoFrecPalabras**: guardaremos en este string el texto una vez ya convertido al leer con el método de leerFrecPalabras.

Clase: ProcesarTexto

- **double[][] matrizFlujos**: aqui 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.

2.2 Descripción 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_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_SimulatedAnnealing:

- **execute()**: Método que llamará a la clase de Simulated Annealing y se irá desencadenando toda la ejecución del algoritmo.
- **getPositions()**: Método que devuelve una matriz Nx2 que representará donde está cada letra colocada una vez se haga la generación del teclado.
- **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.

Clase: A_Branch:

- **escogerDistribucion():** determinará segun 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 cuales son las más cercanas al centro de esta, guardándo 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 i preparará la llamda a branching.
- **branching(level):** funcion recursiva donde segun 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()**: 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.
- **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 trafico 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: P_LeerInput:

- **leerTexto(String nombreArchivo)**: método que lee de un archivo (pasado por parámetro) un texto y lo guarda en el atributo privado texto, antes de leerlo mira que el archivo no esté vacío y que tenga el formato que queremos y si no saltan las excepciones personalizadas nuestras.
- **leerAlfabeto(String nombreArchivo)**: método que lee de un archivo (pasado por parámetro) un alfabeto y lo guarda en el atributo privado de alfabeto, antes de leerlo mira que el archivo no esté vacío y que tenga el formato que queremos sino saltan nuestras excepciones.
- **leerFrecPalabras(String nombreArchivo)**: método que lee de un archivo (pasado por parámetro) palabra junto con su frecuencia (en pares de línea) y lo procesa para juntarlo en un mismo string tantas veces repetida la palabra como su frecuencia y así tratarlo como un texto, antes de leerlo mira que el archivo no esté vacío y que tenga el formato que queremos sino saltan nuestras excepciones.
- **vacioArchivo(String nombreArchivo)**: método que dado el archivo leído por parámetro dice si está vacío o no (boolean).
- **formatoTextoOk(String nombreArchivo)**: método que nos dice si es correcto el formato que queremos del texto que leeremos en leerTexto. Si no contiene numero y solo contiene caracteres como letras y signos de puntuación.
- **formatoAlfabetoOk(String nombreArchivo)**: método que nos dice si es correcto el formato que queremos del alfabeto: una línea de chars seguidos sin repeticiones y en minúsculas y si queremos el espacio lo ponemos un espacio también.
- **formatoFrecPalabras(String nombreArchivo)**: método que nos dice si el formato del archivo que leemos en el método de leerFrecPalabras es el correcto: cada par de líneas que tenga la primera una palabra y la segunda su número de apariciones y por lo que tendrá que ser un número par de líneas al final.
- **procesarFrecPalabras()**: método que devuelve el String de al leer el archivo de frecuencia de palabras, el cual este string tendrá las palabras repetidas tantas veces como su número de frecuencia indica por ejemplo si el archivo contiene (en formato correcto): hola 2 que 4 -> el string será: hola hola que que que que.

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.

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

Arnau:
- A_cotaGilmoreLawler.java
- CotaGLTest.java
- driverGenerarTeclado.java
- P_imprimirTeclado.java
- A_ControladorCapaDominio.java
- D_ControladorCapaDatos.java

Joan Marc:
- A_Branch.java

Pablo:
- P_LeerInput.java
- ProcesarTexto.java
- ExcepFichero.java
- FicheroVacio.java
- FormatoErroneoAlfabeto.java
- FormatoErroneoFrecPalabras.java
- FormatoErroneoMatriz.java
- FromatoErroneoTexto.java

David:
- A_Hungarian.java
- HungarianTest.java
- A_AlgorismeGeneral.java
- A_SimulatedAnnealing.java
- A_QAP.java
- QAPTest.java

4. Descripción Estructuras de Datos

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

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> []**
  Vector que permite guardar la posición {i, j} de los elementos seleccionados para la asignación del coste óptimo.

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.

5. Descripción Algoritmos

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.
   - 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.

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.

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 (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.