Pràctica 5: Diferenciador d'Idiomes

Jaume Adrover Fernandez, Diego Bermejo Cabañas i Joan Balaguer Llagostera

***Resum*—En aquest document s'explicarà el contingut de la pràctica 5 sobre trobar la distància mitjana d'edició de Levenstein entre dos o més idiomes determinats. Explicarem com hem afrontat el problema, quines solucions hem implementat, el codi desenvolupat i finalment unes conclusions amb els problemes que ens hem trobat i algunes reflexions que hem fet.**

1 Introducció

En aquest article explicarem el funcionament i la implementació de la nostra pràctica, estructurant el document de tal forma que es cobreixi la totalitat de aspectes rellevants d'aquesta.

Per una part explicarem l'estructura del MVC que hem relitzat i mostrarem analíticament com funciona aquesta estructura durant l'execució del programa.

A continuació, explicarem breument el problema que ens han proposat resoldre. Mostrarem analíticament les dues solucions que hem implementat per resoldre el problema, juntament amb explicacions en pseudocodi perquè posteriorment es pugui entendre millor el codi que hem implementat

Després, mostrarem el codi implemementat per a resoldre el problema tal i com l'hem explicat a l'apartat anterior.

Finalment, explicarem com funciona la nostra aplicació una vegada s'executa, a més del problemes que ens hem trobat i com els hem acabat resolguent.

2 Implementació del MVC

Una estructura MVC (Model Vista Controlador), consta de 3 grans parts:

- **Model**: és l'encarregat d'emmagatzemar les dades del programa.
- **Vista**: s'encarrega de mostrar la interfície d'usuari amb les dades del model.
- **Controlador**: encarregat de notificar canvis de la vista entre el programa principal i el model de dades.

Una vegada sabem el que simbolitza cada part de l'estructura, podem entrar més en detall en el que fa cada una d'aquestes a la nostra pràctica. Per entendre-ho millor, exemplificarem el paper que jugarà cada una de les parts quan executem el programa.

- Una vegada es posa en marxa el programa principal, podem observar dos components que ens deixen escollir els idiomes a comparar. A l'esquerra tenim la llista dels idiomes implementats i a la dreta, similar però amb l'opció de "tots"afegida.
- Escollim dues opcions i posteriorment, clicam al checkbox si volem utilitzar la versió optimitzada.Aquest checkbox activarà la utilització d'un algoritme més ràpid enlloc del tradicional tots amb tots.
- Una vegada seleccionats tots els idiomes i configuració desitjada, clicam el botó de calcula. El programa principal és avisat mitjançant la vista i el controlador s'encarrega de tota la gestió de les accions.
- Quan el controlador és avisat, aquest comprova si primer hem de comprovar un amb tots o si només son dos idiomes. A continuació, miram si el model té marcada la opció d'utilitzar l'algoritme optimitzat i, finalment, executam els càlculs.
- En darrer lloc, observarem que a la finestra s'escriu el resultat en una etiqueta. Per a poder seguir utilitzant la aplicació, torneu al principi d'aquesta llista i seguiu la mateixa metodologia.

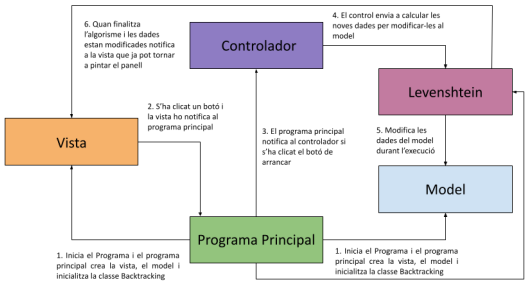

Figura 1. Esquema de com funciona el nostra MVC

3 Implementació de la solució

3.1 Problemes a resoldre

Durant el desenvolupament d'aquesta pràctica, hem tingut diferents inconvenients/dubtes. Vegeu la llista:

- **Qualitat de les Dades**: un dels principals problemes que ens varem trobar va ser amb la qualitat de les dades trobades. Molts de diccionaris eren amb formats extranys

i depenien del país. En resum, la qualitat de la informació no era gaire bóna.

- **Optimització de l'algoritme**: enteníem la manera d'optimitzar, però no teníem molt clar com ordenar les paraules per longitud, a més d'obtenir els indexos de les localitzacions de les paraules segons la seva longitud.

3.2 Solucions adoptades

- **Manipulació de les Dades**: per a tenir unes dades de qualitat, primer de tot, vàrem haver de canviar la codificació a *UTF-8*.Posteriorment, vàrem convertir tot a minúscules amb l'aplicació *Notepad++*. Això vendria a ser els requisits mínims per a la compatibilitat amb la nostra versió de Java, però hem d'afegir aquests si volem utilitzar l'algoritme optimitzat:
  - **Ordenació per longitud**: vàrem aconseguir ordenar por longitud les paraules en el txt que té el següent format: *XXX_sorted.txt*. D'aquesta manera, si només volem comparar paraules de mateixa longitud o no molt diferent a les de una paraula, podem seguir un ordre seqüencial.Destacam que *XXX* es refereix al codi d'un idioma, per exemple, *DEN* en cas del danés.

  - **Emmagatzematge indexos segons la longitud**: aconseguim guardar en un fitxer les respectives posicions on comencen les paraules d'una determinada longitud. Vegeu *metadata* posteriorment.

  Totes aquestes manipulacions s'han aconseguit amb un script de Python, ja que ens resultava més senzill. Vegeu-lo:

  En resum el que es fa a l'algoritme de python és el següent:

  - **Lectura de fitxers i ordenació(línies 4-13)**: com podem observar, es van afegint les paraules a un vector i després s'escriu tot a un fitxer, per exemple *CAT_sorted.txt*, que correspon al català ordenat.

  - **Lectura de fitxers ordenats i metadata(línies 16-35**: en aquest bucle, el que anam fent és llegir els diccionaris ordenats i mitjançant un diccionari anotam a partir de quina posició comencen les paraules d'aquesta longitud. El diccionari segueix aquest format:

    $$dict = \{x_0 : y_0, x_1 : y_1, ..., x_n : y_n | x_i \in [L_0, L_n]\}$$

    D'aquesta manera, sabem que x_i pertany al conjunt de longitud possibles de les paraules d'un idioma (L_i). Per l'altre lloc y_i correspon a l'índex corresponent on comencen les paraules de x_i longitud.
    Per exemple:

    $$dict = \{1 : 0, 2 : 24, 3 : 56, ...\}$$

    En aquest vector, les paraules amb longitud 3 comencen a la posició 56.

Algorithm 1 Ordenació diccionari i generació metadata

```
1:  dicts ← llistaNomsDiccionaris
2:  dicts2 ← llistaNomsDictsOrdenats
3:
4:  for dict in dicts do
5:      i ← 0
6:      for word in dict do
7:          words[i] ← word
8:          i = i + 1
9:      end for
10:     sorted_words ← words.ordena()
11:     fWrite ← dict + str(_sorted.txt)
12:     fWrite.write(sorted_words)
13: end for
14:
15:
16: for dict in dicts2 do
17:     indexos ← {}
18:     pos ← 0
19:     idx ← 1
20:     f ← open(dict)
21:     line ← dict.readline()
22:     while line do
23:         lon ← len(line) - 1
24:         if lon == idx then
25:             pos ← pos + 1
26:         else
27:             idx ← idx + 1
28:             indexos[idx] = pos
29:             pos ← pos + 1
30:         end if
31:         line ← f.readline()
32:     end while
33:     fitxerOut ← dict + str(_metadata.txt)
34:     fitxerOut.write(indexos)
35: end for
```

3.3 Algoritme de Levenstein

Per a calcular la distància d'edició entre dues paraules, farem servir l'algorisme de Levenstein. En el nostre programa farem servir, la versió més comú d'aquest algorisme, la versió recursiva.

Les diferentes operacions que es poden aplicar a una paraula per a editar-la son: reemplaçar, esborrarr i insertar.
Tenint això en compte, poder realitzar crides recursives en les que modificam la paraula per paràmetre i incrementant el comptador del resultat. Aquest comptador ens indicará la distància resultant entre les dues paraules:

- Afegir: $return(a_1..a_{n-1}, b_1..b_m) + 1$
- Esborrar: $return(a_1..a_n, b_1..b_{m-1}) + 1$
- Reemplaçar: $return(a_1..a_{n-1}, b_1..b_{m-1}) + 1$

Farem aquestes crides recursives fins a arribar a un cas base en el que una de les dues paraules queda buida. En aquest cas haurem de retornar la longitud de l'altra paraula, això representa que quan una para es buida l'única opció es reemplaçar amb tots els caràcters de l'altra paraula.

En resum: La complexitat temporal de la solució anterior

$$\operatorname{lev}(a,b) = \begin{cases} |a| & \text{if } |b| = 0, \\ |b| & \text{if } |a| = 0, \\ \operatorname{lev}\left(\operatorname{tail}(a), \operatorname{tail}(b)\right) & \text{if } a[0] = b[0], \\ 1 + \min \begin{cases} \operatorname{lev}\left(\operatorname{tail}(a), b\right) \\ \operatorname{lev}\left(a, \operatorname{tail}(b)\right) \\ \operatorname{lev}\left(\operatorname{tail}(a), \operatorname{tail}(b)\right) \end{cases} & \text{otherwise} \end{cases}$$

Figura 2. Fórmula resum de la distància de Levenstein

és exponencial. En el pitjor dels casos, podríem acabar fent $O(3^m)$operacions. El pitjor cas es produeix quan cap dels caràcters de les dues cadenes coincideix. A continuació es mostra un diagrama de crida recursiva per al pitjor cas. L'espai auxiliar utilitzat és $O(1)$, ja que no s'utilitza espai addicional. Finalment, una altre opció es que si els últims caràcters de les dues cadenes són iguals, no hi ha gaire cosa a fer. Ignora els últims caràcters i obtén el recompte per a les cadenes restants. Així que fem una crida recursiva per a les longituds m-1 i n-1:

Algorithm 2 Distància d'edició entre dues paraules

```
function EDITDIST(str1, str2, m, n)
    if m = 0 then
        return n
    end if
    if n = 0 then
        return m
    end if
    if str1[m − 1] = str2[n − 1] then
        return EDITDIST(str1, str2, m − 1, n − 1)
    end if
    return 1 + min(
          EDITDIST(str1, str2, m, n − 1),          ▷ Insert
          EDITDIST(str1, str2, m − 1, n),          ▷ Remove
          EDITDIST(str1, str2, m − 1, n − 1)       ▷ Replace
    )
end function
```

3.4 Algorisme no optimitzat

Ara volem implementar la solució al problema donat. Comparar dos idiomes europeus, a partir dels seus diccionaris de text. Aquesta primera solució que aportarem es la menys eficient (ho comprobarem més endavant).

Per començar necesitam els fitxers de text que contenen les paraules traduides entre els dos idiomes que volem comparar. Una vegada tenim això hem de carregar els dos idiomes seleccionats a la GUI a la classe mode, ja que l'algorisme accedeix a les dades del model.
La dinàmica per poder resoldre l'algorisme es la següent:

1) Emmagatzemar la primera paraula del primer diccionari y comparar-la amb totes les de l'altre diccionari
2) Mentre iteram per aquest diccionari emmagatzemam calculam la distancia de Levenstein entre la paraula del primer diccionari i la del segon en l'actual iteració.
3) Aquests resultats els anam guardant i comparant amb l'anterior per a obtenir la mínima distància d'edició.
4) En quant hem hem acabat els dos bucles aniats sumam les mínimes distàncies de cada una de les paraules del primer diccionari i la dividim entre el nombre de paraules d'aquest idioma per a obtenir la mitjana de les mínimes distàncies de les paraules.
5) Una vegada hem acabat els dos primers bucles, hem de realitszar la mateixa mecànica pero intercanviant l'ordre dels dos llenguatges.
6) Finalment, quan tenim les dues mitjanes dels dos idiomes hem de retornar el resultat de la següent fórmula que ens indicarà la distància resultant entre els dos idiomes. $\sqrt{(mitjana_1^2 + mitjana_2^2}$

3.5 Algorisme optimitzat

Aquest algoritme presenta una optimització bastant considerable a l'hora de comparar i obtenir la distància mínima. Si ens adonam compte, sols fa falta comparar una paraula amb les altres que tenguin la mateixa longitud, ja que la distància mínima pot ser 0, ja que no haurem d'afegir mai cap lletra per a igualar la seva longitud. Anem a veurer-ho d'una manera molt més analítica:

$$\exists \quad \vec{w}_1, \vec{w}_2, L_1, L_2 \quad | \quad \vec{w}_1 \in L_1 \wedge \vec{w}_2 \in L_2$$

Existeixen dues paraules w_1 i w_2, que ambdues tenen longitud l_i. Si ens adonam compte, el rang de distància sempre segueix aquest interval:

$$Dist_{12} = [|l_1 - l_2|, l_{max}]$$

$$Dist_{12} = Dist_{21}$$

´D'aquestes fórmules extreim dues conclusions:

- **L'ordre no importa**:comparar la distància de la primera paraula o viceversa dona el mateix resultat. Per exemple:

$$\vec{w}_1 = arbre$$

$$\vec{w}_2 = arbres$$

Si tenim aquestes dues paraules, de la primera a la segona hi haurà distància mínima de 1, ja que sempre s'hi haurà d'afegir una lletra. En canvi, per l'altre banda, sempre haurem d'eliminar una lletra per a obtenir la mateixa longitud.

Algorithm 3 Distància entre dos idiomes (No Optimitzat

```
function DISTENTRE2LANGS

    int minDist
    Double mean1 = 0.0
    for i ← 0 to llenguatge1.length do
        str1 ← llenguatge1[i]
        minDist ← ∞
        for j ← 0 to llenguatge2.length do
            str2 ← llenguatge2[j]
            int dist ← EDITDIST(str1, str2, str1.length(),
str2.length())
            if dist < minDist then
                minDist ← dist
            end if
        end for

        mean1 += minDist
    end for
    mean1 ← mean1 / llenguatge1.length

    Double mean2 = 0.0
    for i ← 0 to llenguatge2.length do
        str1 ← llenguatge2[i]
        minDist ← Integer.MAX_VALUE
        for j ← 0 to llenguatge1.length do
            str2 ← llenguatge1[j]
            int dist ← EDITDIST(str1, str2, str1.length(),
str2.length())
            if dist < minDist then
                minDist ← dist
            end if
        end for

        mean2 += minDist
    end for
    mean2 ← mean2 / llenguatge2.length

    return √((mean1 · mean1) + (mean2 · mean2))
end function
```

- **Optimitzar nº comparacions**: d'aquesta manera, podem veure que la millor manera de trobar una distància mínima és on la diferència agafa el valor més petit possible. Aquesta condició s'acompleix quan $l_1 = l_2$, és a dir, quan la longitud d'ambdues paraules és la mateixa. D'aquesta manera, el rang de valor que pot agafar la distància és el següent:

$$Dist = [0, l_{max}]$$

No ho hem mencionat anteriorment pero l_{max} fa referència a la longitud máx de les dues paraules, que és el valor màxim que pot agafar la distància de Levenstein. És a dir, en el pitjor dels casos, totes les lletres de la paraula més llarga són incorrectes i $Dist = l_{max}$.

Com a resultat de l'anàlisi anterior tenim 2 possibilitats diferents:

$$f(x) = \begin{cases} seguim\ mirant & \text{if } D > 1, \\ aturam & \text{if } D \leq 1. \end{cases}$$

És a dir, si trobam distància menor o igual a 1 després d'haver mirat totes les paraules d'una mateixa longitud, no fa falta que seguim mirant les paraules adjacents de altres longituds. Això passa ja que tant per l'esquerra com la dreta les pròximes paraules tindràn sempre distància mínima 1 i no es pot trobar un cas millor.

- **Esquerre**: en el cas de les paraules amb longitud menor, trobam aquest rang de distàncies:

$$l_1 = l_2 + 1$$

$$Dmin = l_1 - l_2 = |l_2 + 1 - l_2 = 1$$

$$Dist = [1, l_1]$$

  D'aquesta manera, en el millor dels casos no aconseguiríem millor la distància mínima actual, així que no fa falta seguir mirant si la nostra distància és menor igual a 1.
- **Dreta**: en el cas de les paraules amb longitud major, trobam aquest rang de distàncies:

$$l_1 = l_2 - 1$$

$$Dmin = l_1 - l_2 = |l_2 - 1 - l_2| = 1$$

$$Dist = [1, l_1]$$

  D'aquesta manera, en el millor dels casos no aconseguiríem millor la distància mínima actual, així que no fa falta seguir mirant si la nostra distància és menor igual a 1.

En canvi, si la nostra distància D és major que 1, hem de cercar un rang de distàncies on la mínima sigui menor que la actual. Anem a cercar aquest conjunt vàlid:

$$\exists D > 1$$

$$Dist = [D_2, l_{max}]$$

$$D_2 = |l_1 - l_2'| < D$$

És a dir, nosaltres volem trobar un D_2 que sigui menor que la nostra distància actual D. Per tant, hem de trobar un valor en el qual la diferència de longitud de les paraules sigui menor a D. És a dir, hem de poder trobar un cas millor al que tenim actualment, sinó no fa falta mirar el rang amb les paraules corresponents.
Suposem un exemple concret:

$$D = 3$$

$$Dist = [D2, l_max]$$

$$D_2 = l_1 - l'_2 < 3$$

En aquest exemple podem veure que, com a màxim, podem comparar amb paraules que tenguin una diferència de longitud de 2 unitats, sinó $D_2 = D$ i no podem millorar.

3.6 Un idioma amb Tots

Per a calcular la distància d'edició d'un idioma amb tots, hem d'executar l'algorisme apropiat N vegades, essent N el nombre d'idiomes.
Cream un array amb els idiomes inicialitzats i també inicialitzam la classe que conté els algorismes. Una vegada fet això comprobam al model si l'usuari ha seleccionat l'opció optimitzada o no. Llavors començam el bucle i les execucions mentre o ammagatzemam a un vector que mostrarem.

4 Model

A la classe model tenim diferentes estructures que explicarem més endavant per a poder emmmagatzemar les dades que necessitarem. Aquestes són:

```
private final String[] dicts;

private Idioma idioma1;

private Idioma idioma2;

public boolean tots;

private boolean optimitzat;
```

- `dicts`: En aquest array contenim els pseudònims que identifiquen als 10 diferents llenguatges. Aquestes dades ens permeten obtenir totes les diferents opcions a l'hora de comparar un idioma amb tota la resta.

- `prog`: Aquesta es una instància de la classe principal.

- `idioma1 i idioma2`: Aquestes son les dues diferentes instàncies dels dos idiomes que compararem a l'algorisme, per tant, haurem de establir els valors al model abans de fer l'algorism , ja que lógicament l'algorisme realitza els càlculs sobre els idiomes que es troben al model. Cal dir que quan feim una execució de un idioma amb tota la resta el segon atribut sirà null i no contindrà cap informació, degut a que haurem d'instanciar tota la resta.

- `solucionat`: valor booleà que ens indica si ja s'ha trobat la solució, sirem notificats.

- `tots`: valor booleà que ens indica si l'usuari vol realitzar una execució d'un idioma amb tota la resta.

- `optimitzat`: aquest valor booleà ens indicarà, en aquest cas, si l'usuari desitja executar l'algorsme optimitzat o no, segons si ha marcat la casella a la Vista o no.

Passem ara a explicar els métodes i funcions del Model són principalment *getters i setters*, com per exemple `getIdioma1();`, que obté el contingut de l'idioma1 que ha seleccionat l'usuari per tal d'obtenir el seu array de paraules i poder executar l'algorisme.

4.1 Idioma

La classe Idioma.java emmagatzema les dades individuals d'un diccionari:

- `List<String> words`: Com el propi nom indica aquest atribut una llista enllaçada de totes les paraules que es troben el fitxer de l'idioma corresponent.
- `List<String> sortedWords`: aquesta es una estructura que conté el contingut de les paraules de l'idioma peró aquest llegeix el fitxer que es troba ordenat.
- `int indexos[]`: Aquest es l'array que conté els indexos de l'array de paraules ordenades on canvia la longitud d'aquestes. Per exemple si les paraules de 3 lletres a l'array de paraules comencen a l'index 30, llavors la tercera posició de l'array d'indexos contindrà el número 30. Aquest array l'inicialitzam llegint dels fitxers continguts a la carpeta metadata. L'utilitat d'aquesta estructura de dades es basa en la facilitat per poder dissenyar l'algorisme optimitzat.
- `String nom`: Com el seu propi nom indica, aquest atribut ens permet obtenir el pseudònim de l'idioma contingut a la classe.

Els mètodes d'aquesta classe son principalment `getters i setters`. Però, a part d'això, tenim els métodes per inicialitzar les estructures de dades: paraules, paraules ordenades i els indexos. Els dos primers es basen en la mateixa mecànica de llegir el fitxer seqüencialment amb un bucle while i anara afegint amb el mètode `.add()`.
En canvi, el mètode per inicialitzar l'array de indexos es un poc diferent degut al propi contingut del fitxer. Les línies del fitxer contenen dos valors que ens aporten informació i venen separats per una ","el primer valor numèric que trobam a una línea del fitxer és la longitud en sí de les paraules, mentre que l'altre conté l'index on es comencen a trobar paraules d'aquesta longitud a l'array de paraules. Per fer l'inicialització, a la lectura seqüència hem d'utlitzar el métode `.split(",")`, a cada nova línea que trobem i guardar els dos valors de l'array que ens retorni el mètode encara que l'únic que ens interesa es l'index que el guardarem a la posició que pertoqui segons la longitud que hagem trobat, d'aquesta manera si l'idioma no té paraules de , per exemple, 14 lletres llavors la posició 14 de l'array d'índexos a la classe `Idioma`, tindrà un contingut null.

5 Vista

La Vista, té com a funció, dintre de l'estructura del programa, mostrar a l'usuari mitjançant un GUI les dades que actualment es troben al model. Els atributs que pertanyen a la classe principal de la `Vista` són el següents:

```
private final Main prog;
private final Panell panell;
JLabel resultLabel;
JComboBox<String> comboBox1;
JComboBox<String> comboBox2;
JCheckBox checkBox;
```

- `prog`: atribut que és una instancia del programa principal. Amb ell, podrem obtenir l'atribut `resultLabel` Aquesta representa la secció de la finestra on mostrarem els resultats obtinguts de l'execució de l'algorisme.

- `panell`: atribut que representa l'objecte `Panell` on dibuixarem tots els punts que generem.

- `combobox1 i combobox2`: aquests son els atributs que contenen els components de selecció múltiple per poder seleccionar els idiomes destí per a realitzar l'algorisme.

- `checkBox`: aqui mostram un marcador per si l'usuari desitja aplicar l'opció de l'algorisme optimitzat.

Pel que fa als mètodes de la classe simplement són 5:

```
public void mostrar();
public void initComponents();
public void actionPerformed(ActionEvent e);
public void notificar(String s);
public void mouseClicked(MouseEvent e);
```

- `mostrar()`: aquest mètode bàsicament crida a totes les funcions encarregades de generar i mostrar les components a pintar

- `afegeixComponents()`: en aquesta funció cream els botons, comboboxes i els inicialitzam amb els valors disponibles al programa. També dotam als components d'un LookandFeel per cambiar l'aspecte per defecte. El marcador o checkbox li afegim un ActionListener que modifica l'atribut booleà del model en cas de que l'usuari vulgui realitzar l'opció optimitzada

- `actionPerformed(ActionEvent e)` : mètode que s'executa quan algun dels botons de la GUI es clica. Emmagatzema el text que està escrit al botó i l'envia com a comanda al programa principal (que el tenim emmagatzemat a l'atribut `prog`).

- `notificar(String s)`: mètode que utilitzam per notificar al control de que l'usuari ha clicat el botó per a calcular .

- `mouseClicked(MouseEvent e)`: En aquesta funció agafarem els botons seleccionats de model per a que es puguin pintar de manera que es distingueixin de la resta.

Els altres mètodes son més simples però igual de rellevants per a la correcta execució del programa. Primer ens trobam dos `getters`, que ens permeten obtenir l'idioma seleccionat als dos comboboxes i un `setter`, que ens posibilita editar el Label per mostrar el resultat desde una altre classe.

5.1 Classe Panell

La classe `Panell.java`, es molt més simple que les vistes anteriorment, ja que no hem d'implementar cap métode per a pintar sobre el panell. Simplement hem de extendre la classe `JPanel`, per a poder establir el tamany preferit que volem per al nostre programa.

6 Controlador

La classe de Control, es l'encarregada de comunicar les opciones seleccionades per l'usuari a la Vista. Si s'ha seleccionat l'opció optimitzada o si es desitja executar un idioma amb un altre s'ha d'agafar aquestes dades de la vista i emmagatzemar directament al model.
En cas de que es vulgui comprobar la distància entre dos únics idiomes, hem de emmagatzemar-los tot dos al model. Una vegada fet aixió els a inicialitzam i executam l'algorisme que l'usuari desitji per a poder mostrar el resultat a l'apartat de la finestra.
L'altre opció es el cas de que l'usuari seleccioni l'opció de calcular la distància d'un idioma amb tota la resta, en aquest cas haurem de fer un bucle executant diverses vegades i emmagatzemant els resultats per a mostrar-los després, tal i com hem explicat abans.
Una vegada calculat únicament haurem de mostrar els resultats a la Vista.

7 Programa Principal

En aquesta secció s'explicarà la estructura que segueix el nostre programa principal, a més de les accions a realitzar. El nostre programa principal conté com a atributs tres diferents instàncies que implementen el MVC:

```
private Model mod;
private Vista vis;
private Control con;
```

Amb aquests punters podem accedir a l'informació del model i notificar el diferents events enregistrats. En quant als métodes, el més important i l'únic digne d'explicació es el de `notificar(String s)` Aquest métode implementa les accions que deriven de l'String rebut per paràmetre, cada posibilitat del switch que utilitzam representa un dels botons de l'interfície les diferentes opcions són:

- `case 'Calcula':` , en aquest botó notificam al control per a que calculi amb el parametre indicat.

En aquesta pràctica únicament tenim un botó per a notificar al Control, pero seguim utilitzant la mateixa plantilla per a simplificar el diseny MVC. Aquesta simplesa a l'hora de notificar, es degut a que l'informació del JComboBox l'agafam directament de la Vista al Model.

8 Joc de proves

En aquest apartat es mostrarán diferents exemples d'execució amb els quals podrem observar el correcte funcionament de l'aplicació.

8.1 Primer cas de prova

En aquest primer cas de prova calcularem la distancia entre els idiomes català (CAT) i espanyol (ESP) utilitzant l'algorisme sense optimitzar. El resultat de l'execució es pot observar a la següent imatge:

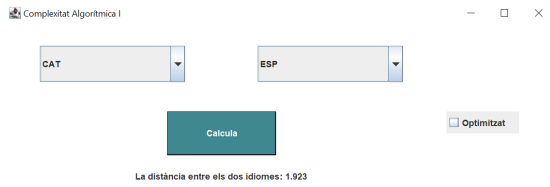

Figura 3. Resultat de l'execució 1

8.2 Segon cas de prova

En aquest segon cas de prova calcularem la distancia entre els idiomes francès (FRA) i anglès (ENG) utilitzant l'algorisme optimitzat. El resultat de l'execució es pot observar a la següent imatge:

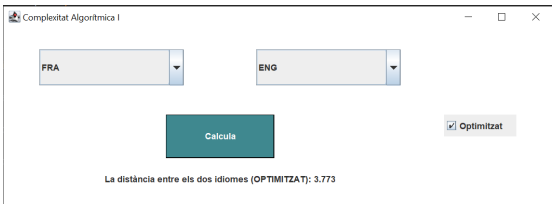

Figura 4. Resultat de l'execució 2

8.3 Tercer cas de prova

Per al tercer cas de prova calcularem la distancia entre els idiomes norueg (NOR) i el suec (SWE) utilitzant l'algorisme sense optimitzar. El resultat de l'execució es pot observar a la següent imatge:

8.4 Quart cas de prova

En el cas del quart cas de prova calcularem la distancia entre els idiomes norueg (NOR) i l'italià (ITA) utilitzant l'algorisme optimitzat. El resultat de l'execució es pot observar a la següent imatge:

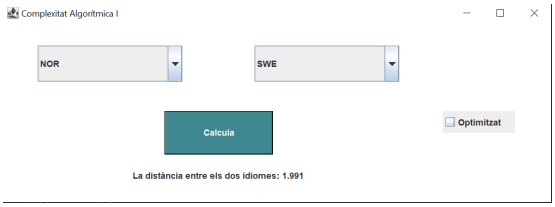

Figura 5. Resultat de l'execució 3

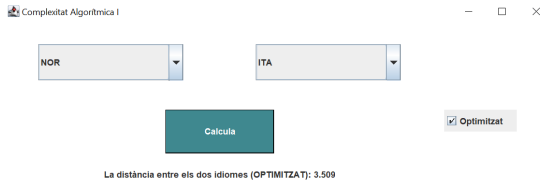

Figura 6. Resultat de l'execució 4

8.5 Cinquè cas de prova

Finalment, farem una execució on es calculi 1 distancia entre una llengua i la resta. En aquest cinquè cas de prova calcularem la distancia entre el català (CAT) i la resta de llengues. El resultat de l'execució es pot observar a la següent imatge, on es mostra la distancia que manté el català amb la resta de llengues al pop up que mostra el programa una vegada l'algorisme acaba de calcular:

Figura 7. Resultat de l'execució 5

9 Comparació d'algoritmes

En aquesta secció, ens encarregarem de comparar en termes de precisió i rendiment els dos algoritmes implementats. D'aquesta manera, hem agafat 5 execucions aleatòries i així veurem quin marge d'error s'aprecia i si el rendiment millora considerablement.

Vegem la següent taula:

	No Optimitzat		Optimitzat	
	Valor	Temps(ms)	Valor	Temps(ms)
CAT-ESP	1'923	860	2'009	396
DEU-ITA	3'671	30.417	3'711	14.142
NOR-POR	3'554	7.141	3'567	5.663
DEN-NOR	1'142	3.958	1'21	1.353
SWE-FRA	3'929	25.156	4'001	11.334

Taula I
VALOR Y TEMPS D'EXECUCIÓ DELS ALGORISMES NO OPTIMITZAT Y OPTIMITZAT

Amb aquests gràfics podem veure diferents fets rellevants:

	Error	Millora
CAT-ESP	4’47	117’17
DEU-ITA	1’09	115’29
NOR-POR	0’37	26’15
DEN-NOR	5’95	192’31
SWE-FRA	1’83	121’89

Taula II
TABLA DE ERROR Y MILLORA

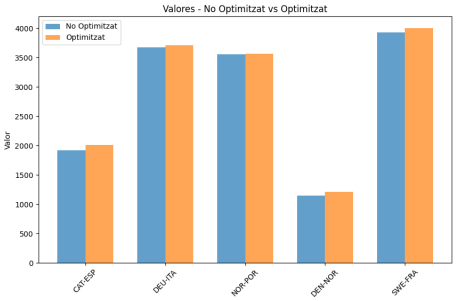

Figura 8. Gràfic mostrant l’error Optimitzat vs No Optimitzat

- Al primer gràfic es mostra el que varem explicar sobre el 2,74% d’error produit entre els dos algorismes. La diferència d’altura entre les dues barres es pràcticament despreciable. Això mostra la correctesa de l’implementació de l’algorisme.
- En el darrer gráfic es mostra la diferència de temps d’execució entre les dues implementacions. Com es pot apreciar, els idiomes amb mayor distància, impliquen un major temps d’execució, per tot dos algorismes. Això es degut a que hem de mirar per més paraules per poder trobar la mínima. Tambe veim una diferencia bastant notória en quant al temps d’execució entre els dos algorismes.

Les fórmules amb que s’han calculat l’error i la millora són les següents:

$$Error = \frac{|V_A - V_B|}{V_A} \tag{1}$$

$$Millora = (\frac{T_A}{T_B} - 1) \times 100 \tag{2}$$

Les variables X_A corresponen als valors de l’algoritme no optimitzat, mentre que X_B corresponen a valors de l’altre algoritme optimitzat.Els resultats signifiquen el següent:

- **Error**: el resultat s’allunya el % determinat de la solució de l’algoritme no optimitzat. Per exemple, si el resultat bo és 100 i ens ha donat 105, tenim un marge d’error del 5%. En el nostre cas, la mitja de l’error segons aquestes execucions es del 2,74 %
- **Millora**: el resultat s’ha calculat un % més ràpid amb l’algoritme optimitzat. Per exemple: un 100% més ràpid significa que ha tardat la mitat i que és dues vegades millor que l’altre.

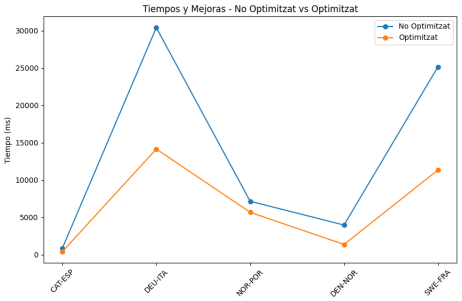

Figura 9. Gràfic mostrant el temps d’execució Optimitzat vs No Optimitzat

Si volem obtenir una mesura més uniforme, ja que tenim valors que canvien molt entre ells, realitzarem la mitjana aritmètica. Hem obtingut els resultats següents:

$$\bar{\epsilon} = 2.742\%$$

$$\bar{\delta} = 114.56\%$$

En conclusió tenim un algoritme que tan sols s’equivoca en un 2.742% de mitja i que és un 114.56% més ràpid, és a dir, que tarda menys de la meitat. Realment és un sacrifici de una petita porció de precisió per un augment molt considerable en el rendiment. Cal destacar que aquest error el provoca el fet de que certs casos no són observats i si ens fixam, el valor sempre tendeix a augmentar en el cas del nostre algoritme. Per tant, suposam que el segon algoritme no contempla certs casos on la distància realment és inferior del que es troba. Finalment, aquests casos que no contempla amb distàncies majors de les originals, es va concatenant i afecta en un percentatge petit degut a la mitjana.

10 Conclusions Finals

D’aquesta pràctica podem extreure conclusions molt interessants. En primer lloc, cal destacar que, malgrat ser més ràpid l’algoritme optimitzat, existeix un petit error. La pregunta és, val la pena la millora de *performance*, sacrificant aquest càlcul més certer amb una imprecisió?

La resposta, com és clar, és sí. En el cas del nostre projecte, el error no repercuteix greument sobre cap factor concret, és a dir, que un error no podria posar en risc la integritat de qualssevol. Si estiguessim en un cas on la precisió és un factor molt important, hauríem de considerar la altra possibilitat. Tot i això, existeix el dilema de que encara que sigui més certer el primer algoritme, hi ha factors externs que podrien alterar el benefici obtingut. Anem a posar un exemple: si tenim un algoritme que tarda un any i falla molt menys que un que tarda minuts, utilitzaríem el de minuts ja que en un any d’execució poden passar moltes coses a nivell de maquinari com caigudes, etc.

Un altre factor a tenir en compte és la heurística que s'ha aplicat per veure la diferència entre dos idiomes:

$$dif = \sqrt{(mean_1)^2 + (mean_2)^2}$$

Aquesta ha estat enginy per part del professor i cal destacar que és útil per a aconseguir que la distància entre dos idiomes no depengui de l'origen, és a dir, que sigui la mateixa distància $A \rightarrow B$ i $B \rightarrow A$. Això ens permet tenir un criteri més correcte i poder definir una uniformitat en els càlculs.

Finalment, cal destacar el poder de tenir les dades ordenades i un índex per a obtenir les respectives posicions de les paraules. Ordenar tot el diccionari i obtenir els índexos no ens considera un gran cost i, si a més d'això, ho deixam per escrit a un fitxer, no haurem de realitzar aquests càlculs cada vegada. Per tant, és molt important destacar que la manera en que estan integrades les dades dins el nostre projecte ens ajuda a obtenir una millora remarcable. Altres mesures com el paral·lelisme o la aleatorietat ens podrien ajudar a augmentar el rendiment. Sobretot, destacar el fet de que ens ha parescut molt eficient el fet de emmagatzemar les dades ordenades per longitud amb metadata, evitant així un procés que es realitzaria cada pic que s'executés el programa.

11 Bibliografia

1) Video Explicatiu de la Pràctica amb execucions
2) https://www.geeksforgeeks.org/edit-distance-dp-5/
3) https://en.wikipedia.org/wiki/Levenshtein_distance