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.

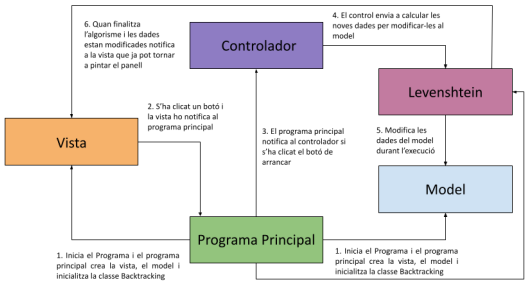

Figura 1. Esquema de com funciona el nostra MVC

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.
- **Problema DARRER**: .

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**:
  - **Emmagatzematge indexos segons la longitud**:

  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.

3.3 Resolució general

3.4 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.

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
```

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: $return(a_1..a_{n-1}, b_1..b_{m-1}) + 1$

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.
La complexitat temporal de la solució anterior 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:

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

4 Model

A la classe model tenim diferentes estructures que explicarem més endavant per a poder emmmagatzemar les dades que necesitarem. 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

6 Controlador

7 Programa Principal

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:

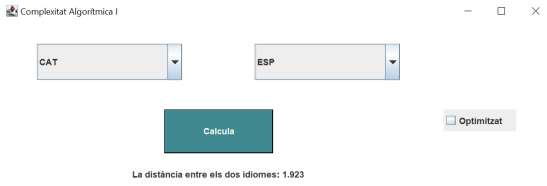

Figura 2. Resultat de l'execució 1

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:

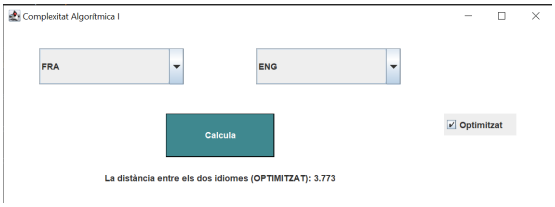

Figura 3. Resultat de l'execució 2

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:

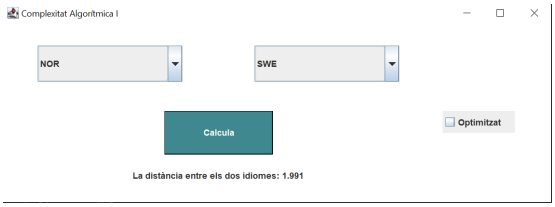

Figura 4. Resultat de l'execució 3

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:

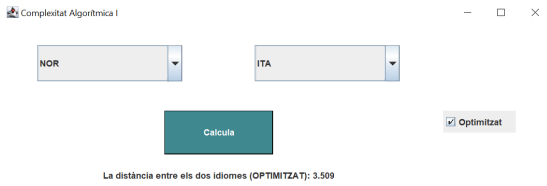

Figura 5. Resultat de l'execució 4

8.5 Cinquè cas de prova

Finalment, farem una execució on es calculi l 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 6. Resultat de l'execució 5

9 Conclusions Finals

10 Bibliografia