Pràctica 5: Diferenciador de diferents idiomes.

Algorismes Avançats - Pràctica 5
(Maig 2023)

———————————————————

Josep Damià Ruiz Pons
Xavier Vives Marcus
Lluis Picornell Company
Martí Paredes Salom

Vídeo explicatiu aquí.

Introducció:

El següent article esdevé una explicació de la quinta pràctica de l'assignatura d'Algorismes Avançats on explicarem l'objectiu, el funcionament i el raonament per aconseguir implementar el projecte.

Aquesta pràctica consiteix en crear un programa que sigui capaç de calcular les semblances d'un idioma enfront d'un conjunt d'idiomes, i obtenir amb quina mesura es sembla més un idioma a un altre.

Per al càlcul de distàncies entre idiomes s'ha emprat l'algorisme dinàmic de Levenhstein vist a classe.

Amés de les funcionalitats obligatòries exposades en l'enunciat de la pràctica el nostre grup ha implementat la funcionalitat de detectar sobre un text donat per l'usuari, de quin idioma es tracta.

Com a patró de programació hem emprat el patró de Model-Vista-Controlador (MVC). Cada part serà desenvolupada més endavant.

MVC

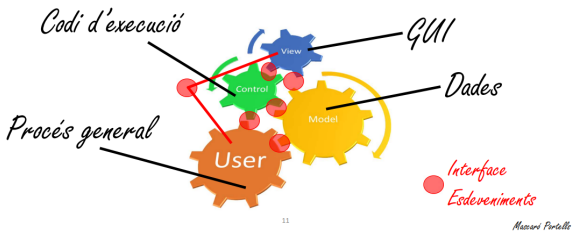
El model vista controlador és una patró d'arquitectura de software que divideix la lògica del programa en tres elements, el Model, la Vista i el Controlador.

Aquests tres elements estan separats de manera interna i cada una s'encarrega d'una tasca diferent i distingida. Ara s'explicaran cada un dels components:

- **Model: El model és l'element o la capa on es treballa amb les dades, és a dir, des de on s'agafarà l'informació, i des de on es podrà actualitzar.**

- **Vista: La vista és l'element o la capa que s'encarrega de mostrar la informació mitjançant una GUI, aquesta informació no és accedida de forma directa, sinó que s'agafa a través de la capa del model.**
- **Controlador: El controlador és l'element o la capa on tota la lògica del programa te lloc. També**

s'encarrega de fer tota la comunicació entre els diferents elements i coordinar-ho tot.
Un controlador només pot tenir una vista, però una vista pot tenir més d'un controlador.



Aquest patró és bastant potent perquè ens permet canviar les diferents components i seguirà funcionant. És a dir, amb aquest patró de programació, podem canviar la interfície gràfica (vista) sense que això afecti al comportament del programa o podríem canviar els algorismes que s'executen (controlador) i només hauriem de canviar aquests, no seria necessari canviar cap altre part del codi.

Hi ha diverses maneres d'implementar aquest model, les dues més comuns són el MVC pur o centralitzat i el MVC per esdeveniments.
El centralitzat com indica el seu nom, tot el flux d'informació passa per el centre o el Main, és a dir, els diferents elements no es poden comunicar entre ells directament. Això fa que tot i que la seva implementació pugui resultar un poc més difícil el manteniment, escalabilitat i la possibilitat de reutilitzar codi és molt major, perquè tot està molt més modulat i es poden reutilitzar cada un dels elements directament.

És a dir, teòricament, un podria reutilitzar els diferents elements del programa i fer-los funcionar en un programa nou amb prop de cap canvi.

El MVC per esdeveniments prescendeix d'aquest enfocament tant modular i permet que els diferents elements es puguin comunicar dirèctament. Això permet eliminar tota aquesta missatgeria que pot parèixer redundant i fer que el procés de creació de codi sigui molt més ràpid, sacrificant totes les avantatges del centralitzat que s'han anomenat anteriorment.

Explicació de les parts de MVC

Per a la nostra pràctica hem adaptat el patró de Model-Vista-Controlador al punt on tenim les dades al model, la interfície d'usuari com a vista i els diferents algorismes al controlador.

Aquestes parts a continuació seran millor desenvolupades però cal aclarir que per realitzar les comunicacions entre els diferents elements, el patró que hem pres és el de per a la part de obtenir informació del model ho feim mitjançant mètodes getter, en canvi per cridar les diferents funcions tant del controlador com de la vista o per actualitzar informació dins del model, empram el mètode clàssic de Model-Vista-Controlador on les diferents parts es comuniquen entre elles fent ús del Main.

En el nostre cas això ho fem mitjançant una interfície de nom "*InterficieComunicacio*"

que implementa un mètode comunicació que rep instruccions i per depenent del tipus de instrucció, crida un mètode o un altre. Per indicar quina instrucció s'ha de realitzar, passam un String que indica el nom de la instrucció i separat per espai hi ha els paràmetres de la instrucció.

Model

El nostre model s'encarrega d'emmagatzemar les dades necessàries per a que el programa funcioni.

En concret el model és l'encarregat d'emmagatzemar els diccionaris i d'implementar la lectura d'aquests fitxers.

- **Diccionaris.**

Els diccionaris és guarden tots dins la carpeta '/Diccionaris/' com a fitxers de text pla amb una extensió '.dic' per a que siguin més fàcils de reconèixer.

Tots els diccionaris contenen a la primera línia el nombre de paraules que contenen, això ens serà molt útil més endavant per aplicar tècniques probabilistiques i reduir el temps de càlcul.

Tots els diccionaris estan codificats amb UTF-8, d'aquesta manera podem comparar idiomes que no siguin de l'alfabet llatí, com el Xinès o el Rus.

- **Model**

Com hem dit anteriorment s'encarrega de llegir els fitxers i carregar els continguts del fitxers pel controlador.

El model té un conjunt de variables privades necessàries per al correcte funcionament:

- **idiomes: Un array de strings amb els idiomes disponibles a analitzar.**
- **diccionarisCarregats: Una estructura hashMap.**
    - **Key (String): L'identificador del diccionari (cat, eng, esp...)**
    - **Values (String[]): Un array amb les paraules del diccionari.**
- **custom: Emprat per emmagatzemar un diccionari creat per l'usuari, ho emprem en l'opció de reconèixer un text proporcionat per l'usuari.**
- **numCPUs: Indicam amb quantes CPUs aplicarem el paral·lelisme de l'algorisme Levenhstein.**

Apart del valors de classe el controlador implementa tota la lògica necesaria per a la lectura dels fitxers '.dic'.

- **carregaDiccionari: Llegeix i emmagatzema a diccionarisCarregats el diccionari amb el nom passat per argument.**
- **addDiccionari: Afegeix un nou diccionar que no es trobi dins la carpeta '/Diccionaris', emprat per transformar el text donat per l'suuari a un diccionari.**
- **getIdiomesCompare: Dona la llista de tots els idiomes a comparar amb el diccionari seleccionat.**

Amés de les mètodes descrits anteriorment i de les variables privades de classe el model compta amb els getters i setter necessaris per garantir encapsulament i privadesa de dades.

Vista

La vista està composta per una finestra (*JFrame)* que compté dues zones:

- **Panell de control**
  - **Zona de botonera. Permet arrancar, aturar reiniciar l'execució i modificar els paràmetres amb els que s'executarà.**
- **Panell dibuix**
  - **S'encarrega de mostrar la solució que el controlador ha calculat. El que mostra serà diferent depenent de quina opció hagi elegit l'usuari.**

Quan l'usuari sel·lecciona el que vol fer (Distància entre dos idiomes, Distància entre un idioma i tots els altres, Arbre filolèxic, Graf de distàncies, Reconeixedor de idiomes), la Vista envia una notificació al controlador a través del Main per especificar l'opció escollida. També ho notifica al Panell Dibuix perque aquest es prepari per mostrar les dades d'una forma determinada.

Representació de dades

- **Distància entre dos idiomes**
  - **Mostra un idioma a esquerra i dreta i una línia amb la distància que els separa**
- **Distància entre un idioma i tots els altres**
  - **Mostra una llista d'idiomes a esquerra i dreta i una línia amb la distància separa cada parella. L'idioma de l'esquerra sempre serà l'idioma principal escollit.**
- **Arbre filolèxic**
  - **Es dibuixa un arbre on les fulles son els idiomes. Conforme anam pujant en l'arbre, es van fusionant fulles fins que només queden dues fulles que formen l'arrel.**
  - **El criteri per fusionar dues fulles es basa en si la distància que les separa és mínima.**
  - **Amb cada fusió de fulles s'ha d'ajustar la distància de la nova fulla fusionada amb totes les altres fulles per tal de que en la següent iteració tengui en compte que aquelles dues fulles s'han fusionat i han passat a ser una sola.**
- **Graf de distàncies**
  - **Es dibuixa un graf complet (tots els nodes connectats amb tots els altres nodes) on cada noda representa un idioma i al punt central de cada aresta s'escriu la distància que separa cada idioma. S'ha arredonit el resultat a només 3 xifres decimals per no sobrecarregar d'informació**
- **Reconeixedor de idiomes**
  - **Apareix una finestra pop-up tipus JDialog amb l'idioma solució i la distància mitja que ha tengut amb el text introduit.**

Optimitzacions fetes

- **Per tal de que la vista no es quedi "penjada", la vista crea un nou fil (*Thread*) anomenat *Dibuixador* que s'encarrega d'actualitzar-la.**

- **Per tal de no estar actualitzant la vista constantment de forma innecessària, el *Dibuixador* només refresca la pantalla quan reb la notificació de que s'ha d'actualitzar. De manera que el *Dibuixador* segueix el següent cicle:**

1. **Dormir fins que rebi una notificació de que hi ha hagut canvis**
2. **Despertar-se i dibuixar els canvis**
3. **Publicar-los a la pantalla**
4. **Tornar al pas 1**

- **Addicionalment, la vista s'encarrega del control de l'aturada del programa. És a dir, quan l'usuari decideixi tancar la finestra o prémer el botó d'aturada, s'aturaran tots els fils d'execució que s'han llençat. Per fer això, s'ha fet que tots els fils que es llencen tenen la següent estructura:**

1. **Comprova si s'ha enviat notificació d'aturada**
   1.1. **(Si s'ha enviat la notificació d'aturada) Atura el fil**
2. **Executa el fil**
3. **Torna al pas 1**

De manera que quan s'enviï l'ordre d'aturada, la condició d'aturada valdrà *TRUE* i s'acabarà el fil.

Controlador

Per finalitzar amb els components del MVC tenim el controlador, que com ja s'ha dit anteriorment és el responsable del procés de càlcul, en aquesta pràctica és la part encarregada de calcular la distància dels diferents idiomes y de tot la lògica necessària per a les funcionalitats extres que hem implementat.

El controlador és un fil en si mateix ja que s'ha de poder seguir emprant el programa mentres el procés de càlcul estigui en marxa. Si no ho fos tota l'aplicació quedaria penjada fins que aquest acabes.

El controlador té un enllaç amb el model i amb el main per a poder comunicar-se directament amb el model de manera directa, o amb el main per després distribuir el missatge a la vista.

Els mètodes del qual disposa el controlador són els següents:

***distanciaTotsIdiome*s: donat un diccionari com a paràmetre es calculen les distancies d'aquest idioma amb tots els disponibles dins del model. Les distàncies és van guardant dins un hashMap, sent la key el nom del idioma i el value la distància que acaba de calcular, aquest hashMap és l'objecte que retorna el mètode.**

***reconeixerIdioma*, donat un text com paràmetre d'entrada crea un nou diccionari**

amb el mètode addDiccionari del model esmentat anteriorment i calcula la distància d'aquest nou diccionari amb tots emprant el mètode *distanciaTotsIdiome*s. Una vegada ha finalitzat el procés agafam l'idioma que té la menor distància, sent aquest l'idioma amb més possibilitats d'encaixar amb el introduït per l'usuari.

***distanciaEntreDosIdiomes*, aquesta funció calcula la distància entre dos idiomes utilitzant l'algoritme de Levenshtein. Per fer-ho, carrega els diccionaris dels idiomes especificats (idioma principal i idioma a comparar). A continuació, fem ús del probabilístic agafant un nombre determinat de paraules (RANDOM_MAX al model) per així alleugerir la càrrega de feina que té l'algorisme. Ja que hem considerat que agafant un 50% de les paraules del diccionari, la distància entre dos idiomes no varia gaire en comparació a lo molt que agilitza l'execució del programa.**

A través de diferents bucles, la funció calcula la distància per a cada paraula de l'idioma principal respecte a les paraules de l'idioma a comparar i viceversa. Utilitza múltiples fils (threads) per a realitzar les comparacions de forma paral·lela i optimitzar el rendiment.

En cada iteració dels bucles, es crea una instància de la classe "Comparador", que compara dues paraules utilitzant la distància de Levenshtein. Cada instància es col·loca en un fil (thread) separat, i s'inicia l'execució de tots els fils generats (el nombre de fils emprats depèn de l'ordinador en s'executi el programa ja que al model s'aconsegueix el nombre de CPUs disponibles mitjançant la funció Runtime.getRuntime().availableProcessors();) .

Un cop s'han completat totes les comparacions en paral·lel, s'obté la distància mínima per a cada paraula dels idiomes principal i a comparar. Això s'aconsegueix agafant els resultats de les instàncies de "Comparador" i actualitzant el valor mínim trobat.

Finalment, es calcula la distància mitjana entre els idiomes principal i a comparar dividint la suma de les distàncies individuals pel nombre de paraules comparades en cada idioma. La distància final es calcula utilitzant la fórmula de la distància euclidiana entre les distàncies mitjanes dels dos idiomes.

La distància calculada s'assigna a la variable "distantcia" de l'objecte "model" i es retorna com a resultat de la funció.

***getMinRes*: donat un hashMap amb els idiomes i la distància relativa a un idioma arbitrari obté la key i el value de la distància menor dins del conjunt. És un mètode molt útil que empram a casi totes les funcions del controlador.**

***distanciadeLevenshtein*: implementació amb Java de l'algorisme de Levenshtein, aquest mètode s'explicara en més detall en la secció de programació dinàmica.**

Cal remarcar que tots el mètodes compleixen amb la privadesa de dades i l'abstracció d'aquestes.

***Classe Comparador*, en aquest codi, la classe Comparador implementa la interfície Runnable, el que permet executar la seva funcionalitat en un fil d'execució separat. L'objectiu principal d'aquesta classe és comparar dues paraules utilitzant l'algorisme de distància de Levenshtein. La classe té les següents parts importants:**

word1 i word2: variables de tipus String que emmagatzemen les paraules a comparar.

res: variable de tipus int que guarda el resultat de la comparació de les paraules.

El constructor Comparador(String w1, String w2): inicialitza les variables word1 i word2 amb els valors passats com a paràmetres.
El mètode run(): s'executa quan la classe és executada com a fil d'execució independent. En aquest cas, crida la funció distanciadeLevenshtein(word1, word2) per calcular la distància de Levenshtein entre les dues paraules i guarda el resultat a la variable res.
El mètode getRes(): retorna el resultat de la comparació de les paraules.
Aquesta classe és útil quan volem comparar les paraules en fils d'execució diferents per aconseguir un processament paral·lel. Cada instància de Comparador pot ser executada com a fil independent i, un cop finalitzada, podem obtenir el resultat de la comparació utilitzant el mètode getRes().

Cost asimptòtic:

Una persona que no sap d'informàtica podría pensar que per tal de saber lo eficient que és un algorisme s'ha de provar, és a dir, que aquest valor s'ha de trobar de manera experimental, cosa que no és certa.

Per tal de poder calcular el cost computacional d'un algorisme de manera teorica tenim el càlcul d'O(n). El fet de poder calcular-ho de manera teorica pot no parèixer una cosa massa important, però això vol dir que podem saber el cost computacional del nostre programa antes fins i tot de començar a picar codi!!!!

Però com funciona? La notació O(n) crea una funció que representa el temps que tardaria un cert algorisme segons la quantitat d'entrades n. D'aquest càlcul es centre en la instrucció crítica, és a dir, la instrucció mes pesada o l'instrucció que farà que l'algorisme tardi més. D'altre banda ignora tot el cost que sigui d'un ordre menor o igual al major, és a dir, si tenim que un conjunt d'instruccions tenen un cost de n2 i unes altres n, aleshores tindrem que tenim un O(n) = n2 + n, però com que estem mirant el cost a l'infinit, aleshores l'n es va fent despreciable, per lo que diem que té un cost de O(n) = n2 + n = n2.
De la mateixa manera si tenim dos conjunts d'instruccions amb el mateix cost, com per exemple n i n, aleshores tindria que O(n) = 2n, però com ja hem dit ignoram els costos que siguin inferiors o iguals, per lo que ens quedaria com a O(n) = 2n = n.

$$\sum_{i=1}^{n-1}(n-i)=\sum_{i=1}^{n-1}n-\sum_{i=1}^{n-1}i=(n-1)n-\frac{n(n-1)}{2}$$

$$\Downarrow$$

$$O(n^2)$$

$$\text{obvi}: \Theta(n^2); \text{instrucció crítica}$$

Tot i que és cert que el cost asimptòtic ens dóna una idea de com es comporta un algorisme per a n's molt grans, per això es diu càlcul asimptòtic, perquè es va tornant més precís com més a prop de l'infinit està.

I això que vol dir?, Idò que amb 'N's relativament petites no ens diu com es comporta, això vol dir que un algorisme amb un O(n) = n3 podría ser més ràpid que un de O(n) = n2.
Però això com pot ser possible?, idò per la mateixa manera per la que el càlcul d'O(n) és tan senzill, perquè estem aproximant moltes coses. Tot això ens pot dur a tenir una x molt gran multiplicant al nostre O(n), aquesta x es diu constant multiplicativa.

Tornant al cas anterior posem que el primer algorisme te un T1 = 3n3, i el segon
T2 = 600n2, aquests dos costos son en realitat T1 = O(n) = n3 i T2 = O(n) = n2.
Però si ens fixam és obvi que amb n's petites el T1 és més ràpid que el T2, i si fem una equació molt fàcil, obtenim que l'algorisme 1 és millor que el 2 per valors de n<200.

- **Consideracions generals. 1:**
  - El criteri asimptòtic és una eina molt útil per fer primeres aproximacions.
  - Cas concret:

$$\left.\begin{array}{l} T_1 = 3n^3 \\ T_2 = 600n^2 \end{array}\right\} \text{ és obvi que } \Theta(n^2) \subset \Theta(n^3)$$

  - A la pràctica l'algorisme 1 és millor que el 2 per valors

$$n < 200$$

Però això què vol dir?, idò molt fàcil, que nosaltres com a programadors hem de saber i triar quin és el millor algorisme per cada problema. Ja que un algorisme podria ser molt bo amb n's grans però molt dolent amb n's petites. Per lo que depenet del cas pràctic per el qual estem programant s'haurà d'elegir un algorisme o un altre.

Una altre consideració que s'ha de tenir a l'hora de programar, és el temps de programació i el fet de que a vegades no val la pena optimitzar un programa. Per exemple si amb l'algorisme més lent ja estem dins el temps de resposta que ens demanen no fa falta perdre més temps cercant una millor solució.

També hem de considerar el temps de programació, és a dir, si per programar un algorisme T tarda una setmana i per un algorisme X triga tres mesos i l'X només és 2 segons més ràpid amb un temps total d'una setmana d'execució. Si aquests dos segons no son crítics, com a programadors podem decidir s'utilitza l'algorisme més dolent però guanyant molt de temps de programació.

Cost asimptòtic del problema

El cost computacional de l'algorisme de Levenshtein be condicionat per les dues cadenes de caràcters de l'entrada, amb les quals ha de fer una matriu on emmagatzema tots els resultats intermitjos. Com que ha de fer la matriu i s'ha d'emmagatzemar aleshores té un cost asintotic temporal de O(m*p) que és el mateix que el cost asintotic espacial.

Tot i que utilitzant les tècniques de dividir i vèncer fem que no hagi de fer recalculs innecessaris ja que ja els tindrem guardats.

Després hem de calcular el cost de calcular cada paraula de cada llenguatge amb la resta

de paraules de cada llenguatge, això és òbviament és un O(n2).

Per lo que a nosaltres ens quedaria un O(n2*p*m) , però com que en el nostre cas sabem que ni p ni m seran mai majors de 40, que és un número relativament petit, aleshores ho podem simplificar a O(n2).

Programació dinàmica.

La programació dinàmica és una estratègia de resolució de problemes que consisteix en descompondre aquests problemes amb problemes més petits. El punt més important és que a mesura que va descomponent el problema amb problemes més petits va guardant els resultats, per tal de posteriorment poder-los reutilitzar i no haver de tornar a calcular-los.

El problema dels algoritmes de D&C era que quan es trobava un problema on les sub-solucions no són disjuntes es repetien un munté de càlculs. Aquest problema és resolt amb la programació dinàmica, ja que va guardant els resultats calculats.
Gràcies a això som capaços de resoldre problemes que d'altre manera serien totalment inpracticables pel seu cost computacional.

Però també té certs problemes:
-Hi ha certs problemes, que degut a la grandària dels seus estats, es tornen inpracticables degut al tamany de la memòria requerida per tal de guardar tota la informació requerida . És a dir, que reduint el cost computacional asintòtic d'un algorisme pot derivar a un cost espacial asintòtic impracticable en la pràctica.
-També s'ha de tenir en compte que moltes vegades aquesta tècnica necessita de identificació de subproblemes i del disseny de la funció recursiva i de guardat de resultats, coses que fan que hi hagi bastanta complexitat a l'hora de programar-ho.
-Finalment no sempre es pot utilitzar aquesta técnica, per tal de que es pugui utilitzar s'han de complir dues condicions:

1. La solució total ha de constar d'una sèrie de decisions percials, les quals s'obtenen en una serie consecutiva de pases.
2. Aquesta serie de decisions parcials ha de complir en tot moment el principi d'optimalitat.

Per exemple no es pot resoldre el problema de trobar el camí màxim entre dos vèrtexs d'un graf.

Aquests algorismes són d'especial utilitat en la solució de problemes d'optimització, on s'intenta trobar un màxim o un mínim (principi de Bellman).
Aquest principi diu que donada una seqüència òptima de decisions, tota subseqüència d'aquesta, és a la vegada òptima.

L'esquema general d'una solució trobada a partir de PD és:

1. Dissenyar una seqüència de decisions que complin en tot moment el principi d'optimalitat.
2. Colocar aquestes decisions en un plantejament recursiu.
3. Calcular cada decisió en base a les anteriors, que es troben a memòria, amb el fi de no repetir càlculs.

4. La solució final es recupera desfent el camí emmagatzema en les decisions parcials.

Implementació de la solució:

El codi implementa un algorisme de qualificació de semblances i diferències del tipus dividir i vèncer. En concret utilitza l'algorisme de la distància de edició (Levenshtein).

Aquest algorisme funciona, tal que donades dues seqüències de caràcters X i Y y una serie d'operacions, cada una amb un cost associat, determinar quin és el cost mínim de passar de X a Y aplicant les operacions permeses.
El cost es calcula com el nombre d'operacions que s'han de realitzar.
Les operacions disponibles són replace, delete i insert.

Primer hem de plantejar la solució recursiva de Levenshtein, començam així:
Tenim les paraules a (a1..an) i b (b1..bm) i la seva distància és d(a1..an, b1..bm).
Si la seva darrera lletra és la mateixa (an = bm) aleshores d(a1..an, b1..bm) = d(a1..an-1, b1..an-1), és a dir, la seva distància és la mateixa que sense les seves respectives darreres lletres.
Però si les seves darreres lletres no són iguals (an != bm), aleshores s'ha d'afegir un al cost que ja tenim.

En resum si les lletres an i bm coincideixen aleshores cridam recursivament la funció amb a1..n-1 i b1..m-1 i tornam a evaluar les noves darreres lletres.

Sinó tenim tres opcions, afegir una lletra d(a1..an-1, b1..m), eliminar una lletra d(a1..an, b1..m-1) i modificar una lletra d(a1..an-1, b1..m-1).
Agafam l'opció la qual tingui un cost menor i li sumem un, ja que és el cost de l'operació que hem realitzat.

Els casos trivials son que no hi hagi lletres ni a a ni a b, que aleshores tenen una distancia de 0, també si a no te lletres aleshores la distancia de a i b és la mateixa que la de a i b1..bm-1 +1, és a dir, si una de les dues paraules no te lletres la distància és la llargària de l'altre.

Una vegada s'ha arribat a un cas base es van retornant els costs fins tornar la solució completa.

A mesura que es va pujant desde els casos trivials es va creant una matriu que conté els resultats de les operacions que ja hem realitzat i les distàncies entre totes les sub paraules a i sub paraules b.
És a dir, per cada nivell que es puja aumenta en un el número de files i de columnes amb la nova informació obtinguda en aquell nivell.

		a	l	t	r	u	i	s	t	a
	0	1	2	3	4	5	6	7	8	9
a	1	0	1	2	3	4	5	6	7	8
l	2	1	0	1	2	3	4	5	6	7
g	3	2	1	1	2	3	4	5	6	7
o	4	3	2	2	2	3	4	5	6	7
r	5	4	3	3	2	3	4	5	6	7
i	6	5	4	4	3	3	3	4	5	6
s	7	6	5	5	4	4	4	3	4	5
m	8	7	6	6	5	5	5	4	4	5
e	9	8	7	7	6	6	6	5	5	5

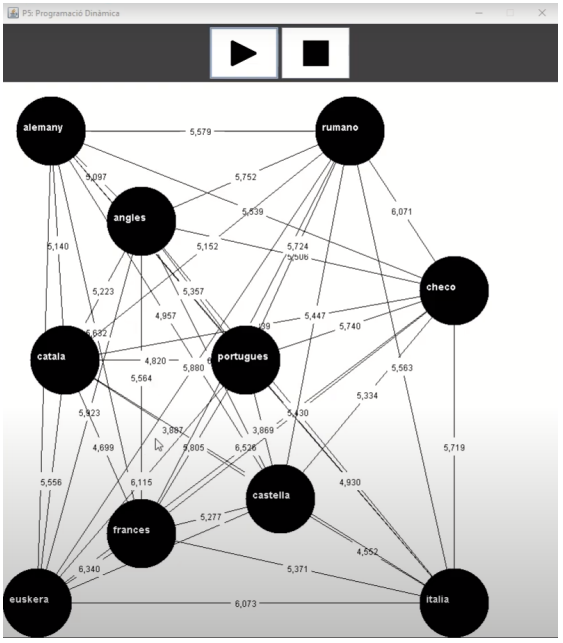
I és aquesta és la matriu que conté tota la informació que s'utilitzarà per tal de no

repetir càlculs i que és el que caracteritza la programació uwu

Guia d'usuari:

En aquesta pràctica l'ús del programa és molt senzill. Consisteix en una finestra composada per un panell on es mostren les diferents distàncies dels diferents idiomes i un panell de control a la part superior de la finestra que oferirà el control del programa:

- **Play: Mostra un segon panell amb totes les aplicacions descrites anteriorment.**
- **Una amb un, l'usuari selecciona dos idiomes i es mostra la distància entre ells i el temps que s'ha trigat pel càlcul.**
- **Tots amb tots, selecciona el llenguatge i s'obtenen totes les distàncies als altres idiomes.**
- **Arbre filolèxic, és mostra l'arbre filolèxic dels idiomes disponibles, és un procés de càlcul llarg ja que és necessari fer les distàncies de tots amb tots.**
- **Graf de distàncies, és una altre representació de tots amb tots, on els nodes son els idiomes i les arestes les distàncies entre els diferent idiomes.**
- **Reconeixedor linguistic, demana a l'usuari un text per a reconèixer i mostra amb un pop-up l'idioma amb més similitud.**



A diferència d'altres vistes que hem fet a altres pràctiques no fa falta seleccionar l'opció a calcular i després prémer start, sinó que una vegada l'usuari selecciona una opció el controlador ja comença a fer els càlculs pertinents a l'opció escollida.

Conclusió:

En conclusió, el problema consisteix en la comparació de idiomes emprant la programació dinàmica i l'algorisme de la distància de Levenhstein.

Per realitzar aquesta pràctica hem hagut d'aprendre com emprar la programació dinàmica i els seus avantatges. Però no només això, ja que hem volgut introduir una sèrie de millores a l'entrega.

La primera millora ha estat la de la paral·lelització de la comparació de paraules. Ja que la càrrega de feina que suposava carregar totes les paraules d'un idioma amb totes les de l'altre era molt gran, el que hem fet ha estat, mitjançant Threads, comparar les paraules per blocs, és a dir si l'ordinador que executa la pràctica té 5 processadors disponibles, comparem les paraules de cinc amb cinc fins que s'acabessin les paraules (cal resaltar que alomillor els grups de paraules no coincideixen amb la divisió de paraules/processadors per lo que el darrer grup era del mòdul de n paraules del diccionari amb m processadors disponibles).

Això redueix considerablement el temps de comparació de dos idiomes i cosa que ens beneficia ja que per a altres funcionalitats de la pràctica com comparar tots els idiomes amb tots el temps d'execució es disparava.

La segona millor implementada ha estat l'ús del probabilístic. En aquesta pràctica hem pegat una primer ullada al tema 7 emprant l'estadística per millorar el temps d'execució.

El que s'ha fet ha estat en lloc de comparar el 100% del diccionari, s'ha carregat un 50% del diccionari i només s'ha comparat aquesta proporció del diccionari.

La següent taula mostra una comparació entre la comparació entre Català i Castella emprant la primera millora i amb la segona enfront a no emprar la segon:

Temps Probabilistic		Sense Probabilistic
Temps (ms)	Distància	Distancia: 1.922
1979	2.921	4772
1607	2.795	4489
1747	2.950	4486
1610	2.770	5869
1607	2.736	5899
2126	2.694	4536
Temps mitjà i distància mitjana		
1778	2.811	5008

S'han realitzat 6 execucions i podem veure com la millora de temps és de 2,8 pics més ràpida emprant el mètode probabilistic.

S'ha de tenir en compte que degut a que s'agafen paraules aleatòries la distància és diferent a cada execució, en comparació a no emprar el mètode probabilístic que sempre és la mateixa, encara que la diferència entre execucions no és massa gran.

El que sí varia és la distància mitjana de les execucions amb la real sense el probabilístic. Això és degut a que com s'agafen menys paraules la comparació de idiomes es torna menys precisa. Si es volgués un resultat més

precís faria falta emprar més paraules al random o emprar un diccionari més complet.

Finalment, també hem inclós l'extra de la identificació d'un idioma mitjançant un text. Això ho hem implementat demanant a l'usuari un text i separant les paraules com si fossin un nou diccionari. Llavors es compara aquest nou diccionari amb tots els altres i l'idioma que tingui distància mínima, serà el que més es pareix al text introduït.

La realització d'aquesta pràctica ens ha permès experimentar amb distintes tècniques d'optimització i aprendre en quines situacions és més convenient emprar-les.