; revised .

Càlcul de diferencies entre llenguatges utilitzant algoritmes de distància.

Dylan Canning Garcia[1], Antonio Marí González[2], Juan Marí González[3], Antoni Jaume Lemesev [4]

[1]Estudiant Enginyeria Informàtica UIB, 49608104W
[2]Estudiant Enginyeria Informàtica UIB, 46390336A
[3]Estudiant Enginyeria Informàtica UIB, 46390338M
[4]Estudiant Enginyeria Informàtica UIB, 49608104W

Alumne d'entrega: Dylan Canning (email: dcg750@id.uib.eu).

ABSTRACT La detecció quantitativa de semblança entre llenguatges és clau en aplicacions que van des de la filologia computacional fins al data cleaning. Aquest treball presenta LexDistance, una aplicació MVC escrita en Java 23 i JavaFX que calcula i visualitza la distància lèxica entre més de deu idiomes utilitzant quatre mètodes clàssics: *Levenshtein*, *Damerau–Levenshtein*, *Longest Common Subsequence* (LCS) i *Jaro–Winkler*. Els diccionaris s'importen automàticament a partir de fitxers `.dic`, `.txt` i `.csv`, i les cadenes es normalitzen en NFC per garantir la coherència Unicode.

El motor de càlcul implementa un patró fork/join sobre un ExecutorService dinàmic que reparteix la generació de la matriu de distàncies en subtasques balancejades; això aconsegueix fins a un 42 % de reducció del temps d'execució en equips quad-core respecte a la versió seqüencial. Els resultats es mostren a la interfície mitjançant una taula interactiva, un graf de semblança i un dendograma generat amb UPGMA. Un servei de notificacions desacoblat sincronitza, en temps real, el progrés de càlcul amb la vista sense bloquejar el fil de JavaFX.

L'avaluació empírica amb cossos lèxics de 50 000 paraules per idioma demostra que Levenshtein i Damerau-Levenshtein ofereixen la millor resolució per a errors ortogràfics, mentre que Jaro–Winkler resulta més indicat per a noms curts. El codi modular facilita l'extensió a nous algorismes i permet exportar els resultats en CSV per a anàlisi posterior.

I. Introducció

AQUESTA pràctica s'ha desenvolupat mitjançant una metodologia estructurada basada en l'arquitectura Model-Vista-Controlador (MVC) vista a classe. La modularitat que presenta aquesta estructura ens ha permès distribuir la feina en paquets d'esforç disjunts, la qual cosa ha agilitzat molt el desenvolupament de la pràctica. Fent ús de les incidències apreses a les pràctiques anteriors, hem modularitzat una mica més la part del MVC.

La distribució de tasques entre el nostre equip de quatre membres s'ha realitzat seguint els components propis del patró MVC [10]: un membre s'ha encarregat de la implementació de la interfície gràfica (Vista), dos membres han desenvolupat la part de dades i algorismes (Model), i un altre ha implementat la part controladora que coordina les interaccions entre els components anteriors.Continuant amb els esquemes anteriors, hem seguit utilitzant el "Factory Method".

Aquest patró de creació encapsula la lògica d'instanciació d'objectes en una classe especialitzada (en el nostre cas, la classe AlgorithmFactory). Això permet que l'usuari (en aquest cas) no hagi de conèixer els detalls de creació dels objectes, ja que aquesta responsabilitat es delega a mètodes específics que, basant-se en el paràmetre com el tipus d'algoritme, generen dinàmicament la instància apropiada. D'aquesta manera, es garanteix una separació clara entre la definició de l'estructura general i les seves implementacions concretes, cosa que afavoreix la modularitat en el nostre cas per fer més dinàmica la creació d'algorismes nous per al nostre codi.

A més, del "Factory Method"hem definit la classe "AlgorithmType "que representa els tipus d'algorismes de càlcul de distàncies, el que permet tenir una definició igual per a tots els algorismes disponibles, cosa que facilita la selecció i la comparativa entre aquests de manera més senzilla. Aquestes dues definicions ens han permès tenir una metodologia més àgil i independent..

Per facilitar la col·laboració i el seguiment del projecte, hem seguint fet ús de l'entorn de treball ja utilitzat a les diverses practiques (repositori GitHub) per a dur un control de versions, permetent documentar i compartir els avanços de manera sistemàtica. Addicionalment, hem seguit fent ús de l'eina *Obsidian* que permet dibuixar diagrames i crear relacions entre idees i arxius. Igualment, amb la mateixa justificació que a les dues entregues anteriors, hem seguit fent feina amb l'ús de l'IDE IntelliJ per mantenir la compatibilitat dels formats de projecte.

Un cop establertes aquestes bases metodològiques, hem procedit a definir els aspectes tècnics específics de cada secció (Algorismes, Estructures de Dades, Processos, etc.) necessaris per a la implementació. L'objectiu central del nostre disseny ha estat aplicar i maximitzar els coneixements adquirits a l'assignatura, especialment en relació amb les estratègies de Programació Dinàmica, Concurrència i Disseny Modular, que constitueixen els pilars per a la facilitat d'implementació de diversos algoritmes que hem investigat.

II. Metodologia de feina

Per aquest projecte hem seguint amb una metodologia estructurada i col·laborativa, que ens ha permès integrar diferents eines i tècniques per aconseguir un producte final coherent sense haver de fer intensius presencials amb tots els companys. En aquest apartat detallarem els aspectes relacionats amb l'entorn de programació i el sistema de control de versions que hem emprat al llarg del desenvolupament, com els canvis en respectiva a la primera pràctica.

A. Entorn de Programació

El desenvolupament de l'aplicació l'hem dut a terme utilitzant l'IDE *IntelliJ IDEA*, que ens ha proporcionat un entorn robust i modern per a programar en Java, bastant més amigable que *NetBeans*. S'ha treballat amb la versió *Java 23*, aprofitant les seves millores i estabilitat per garantir una execució eficient dels càlculs i processos implementats, com també la facilitat de maneig amb objectes gràfics.

Per a aquesta pràctica, hem fet ús de JavaFX per a la interfície gràfica d'usuari per mostrar l'arbre filogenètic i el graf. En concret, hem utilitzat els diversos paquets que ofereix JavaFX per a la representació d'escenes, permetent realitzar els gràfics corresponents en dues i tres dimensions. A més, hem incorporat la interacció amb l'usuari per tal de facilitar la manipulació i exploració dels núvols de punts de manera intuïtiva i dinàmica.

B. Control de Versions

Hem seguit amb l'estructura que varem implementar a la segona pràctica degut a que va funcionar molt bé, es a dir centralitzat al repositori principal on s'allotgen les dues practiques anteriors però seguint amb el nostre sistema de control de versions basat en **Git**, amb el repositori centralitzat allotjat a **GitHub**. Aquesta estratègia ha facilitat:

- Mantenir un historial detallat de totes les modificacions realitzades al codi font.
- Coordinar el treball col·laboratiu entre els membres de l'equip, permetent la integració de diferents funcionalitats de manera ordenada.
- Assegurar una revisió contínua del codi i una resolució ràpida dels conflictes que sorgeixin durant el desenvolupament.

La utilització de GitHub es clau per a la nostra gestió de les versions, degut que ens garanteix que el projecte romangui coherent i actualitzat en tot moment, cosa que afavoreix molt el desenvolupament. A més de l'esmentat, la IDE emprada ens ha permès dur a terme aquest control de versions de manera senzilla pels integrants que no han emprat abans la feina, ja que posseeix eines de gestió de versions sense haver d'utilitzar la terminal.

III. Fonaments Teòrics

En aquesta secció tractarem els diversos elements teòrics que hem emprat per a la nostra pràctica.

A. Programació Dinámica

En l'àmbit de la informàtica, molts problemes d'optimització i anàlisi de dades complexes presenten solucions recursives que, si s'implementen de manera directa, resulten computacionalment ineficients. La programació dinàmica (PD) és una tècnica poderosa per abordar aquests reptes, especialment quan les subsolucions d'un problema se solapen, permetent emmagatzemar resultats intermedis i evitar càlculs redundants. Aquesta tècnica es fonamenta en el principi d'optimalitat de Bellman, que estableix que tota subseqüència d'una solució òptima també ha de ser òptima.

Un dels algoritmes més representatius que es beneficia de la programació dinàmica és l'algorisme de Levenshtein, també conegut com a distància d'edició. Aquest algorisme calcula el nombre mínim d'operacions (insercions, eliminacions o substitucions) necessàries per transformar una cadena de text en una altra. Té aplicacions molt diverses: des de correctors ortogràfics fins a l'anàlisi genètica o lingüística.

En aquesta pràctica es proposa el desenvolupament d'una aplicació amb interfície gràfica d'usuari (GUI), estructurada segons el patró de disseny Model-Vista-Controlador (MVC), que permeti comparar lèxicament un conjunt d'almenys 10 idiomes. L'objectiu és determinar el grau de semblança entre ells utilitzant la distància de Levenshtein ponderada per la longitud de les paraules.

El procés de comparació es basa en calcular, per a cada paraula d'un idioma, la distància mínima respecte a totes les paraules d'un altre idioma. Aquesta distància es mitjana i es combina amb la distància inversa (de l'altre idioma cap al primer) per obtenir una mesura simètrica de semblança. El resultat és una matriu de distàncies entre tots els idiomes, que es pot visualitzar i analitzar des de la interfície.

Aquest enfocament no només permet respondre preguntes com ara "A quina distància està l'idioma X de l'idioma

Y?", sinó que també obre la porta a extensions com la representació gràfica del graf de distàncies, la generació d'arbres filo-lèxics o fins i tot la detecció automàtica de l'idioma d'un text.

B. Patró Factory Method

El patró *Factory Method* és un disseny de creació que hem decidit emprar pel fet que permet delegar la instanciació d'objectes a subclasses, això proporciona una modularitat a l'hora de implementar nous algoritmes. En el context d'aquesta aplicació, hem implementat aquest patró mitjançant la classe AlgorithmFactory, encarregada de generar instàncies d'algorismes en funció dels paràmetres proporcionats.

La interfície comuna `DistanceAlgorithm` defineix el contracte que han de seguir totes les implementacions d'algorismes. Aquesta interfície estableix les operacions `calculateDistance(Word word1, Word word2)`, `calculateNormalizedDistance(Word word1, Word word2)`, `getName()` i `getDescription()`.

A continuació, es mostra el diagrama de classes que exemplifica aquesta arquitectura:

C. Model

La classe AlgorithmFactory és responsable de crear instàncies dels diferents algorismes de càlcul de distàncies disponibles al projecte. Aquesta classe utilitza el patró de disseny "Factory Method"per encapsular la lògica de creació, permetent que el codi client només necessiti especificar el tipus d'algorisme que vol utilitzar, sense preocupar-se pels detalls d'implementació.

Pel que fa als algorismes, el tipus Levenshtein calcula la distància mínima d'edició necessària per transformar una cadena en una altra, considerant operacions com insercions, eliminacions i substitucions de caràcters. És un algorisme clàssic i àmpliament utilitzat per mesurar similituds entre cadenes.

L'algorisme Damerau-Levenshtein amplia el Levenshtein afegint la capacitat de considerar transposicions de caràcters adjacents com una única operació. Això el fa més adequat per a casos on els errors de transposició són comuns.

El tipus Jaro-Winkler dona més pes als prefixos comuns entre dues cadenes, sent especialment útil per comparar noms propis o paraules en idiomes relacionats. Aquest algorisme és ideal per a aplicacions on els prefixos tenen una importància significativa.

L'algorisme Longest Common Subsequence (LCS) busca la seqüència més llarga de caràcters que apareixen en el mateix ordre en ambdues cadenes, sense tenir en compte insercions o eliminacions. És útil per identificar patrons compartits entre cadenes.

Per altra part, la carpeta dictionary se encarga de gestionar la manipulació i càrrega de paraules des de fitxers externs. Aquesta funcionalitat és essencial per proporcionar les dades necessàries per als càlculs de distàncies entre paraules. Els fitxers carregats solen ser de text pla, on cada línia representa una paraula que es processa i es converteix en una instància de la classe Word.

La classe LexModel actua com el nucli del model en l'arquitectura del projecte, integrant i gestionant els diferents components relacionats amb el càlcul de distàncies entre llenguatges. És responsable de coordinar l'ús dels algorismes de distància, la matriu de distàncies i la generació d'estructures derivades com grafos i arbres filogenètics.

Pel que fa als algorismes, LexModel permet seleccionar i aplicar diferents mètodes de càlcul de distàncies, com Levenshtein, Damerau-Levenshtein, Jaro-Winkler, entre d'altres. Aquests algorismes s'utilitzen per calcular la similitud entre llenguatges o paraules, i els resultats s'emmagatzemen en una matriu de distàncies (DistanceMatrix), que és gestionada directament per aquesta classe.

A més, LexModel centralitza la lògica d'esdeveniments i notificacions, publicant actualitzacions sobre el progrés del càlcul, errors o la disponibilitat de dades com el grafo o l'arbre filogenètic. Això la converteix en la classe que "junta tot"en el model, connectant els algorismes, les dades i les notificacions perquè la resta de l'aplicació pugui interactuar amb ells de manera estructurada.

El procés de càrrega d'arxius inclou diverses etapes, com la validació del format del fitxer per assegurar que només es carreguin dades vàlides. Això permet evitar errors durant l'execució del programa. Un cop carregades, les paraules es guarden en estructures de dades adequades per facilitar la seva manipulació i ús en els càlculs.

A més, aquesta carpeta també pot incloure funcionalitats per gestionar diccionaris personalitzats, permetent afegir, eliminar o modificar paraules segons les necessitats de l'usuari. Això fa que el sistema sigui flexible i adaptable a diferents contextos d'ús.

D. Vista

La classe DendogramaView és responsable de representar visualment un dendograma, que és un arbre jeràrquic utilitzat per mostrar relacions entre elements, com llenguatges o paraules, basant-se en les distàncies calculades. Utilitza la biblioteca JavaFX per renderitzar l'arbre, emprant nodes i línies per connectar els elements. Aquesta classe s'encarrega de gestionar el disseny del dendograma, assegurant que les branques i nodes es distribueixin de manera clara i llegible en la interfície gràfica.

La classe LexView actua com la vista principal de l'aplicació, proporcionant la interfície gràfica perquè l'usuari interactuï amb les funcionalitats del model. Utilitzant JavaFX, aquesta classe organitza els diferents components visuals, com menús, botons i panells, per permetre a l'usuari carregar diccionaris, seleccionar algorismes i visualitzar resultats com matrius de distància, dendogrames o grafos. És el punt central d'interacció entre l'usuari i l'aplicació.

D'altra banda, la classe PolygeneticGraphView s'encarrega de mostrar un grafo filogenètic, que representa les

relacions entre llenguatges o paraules en forma de xarxa. Amb JavaFX, aquesta classe utilitza nodes i arestes per construir el grafo, permetent a l'usuari explorar visualment les connexions i distàncies entre els elements. A més, pot incloure funcionalitats interactives, com fer zoom, arrossegar nodes o ressaltar connexions específiques, per millorar l'experiència de l'usuari.

E. Controladora i Notificacions

La classe LexController és l'encarregada de coordinar la interacció entre el model (LexModel) i la vista (LexView) dins l'arquitectura de l'aplicació. Actua com a intermediari gestionant la lògica de l'aplicació, inicialitzant les dependències necessàries i responent a les accions de l'usuari, com carregar diccionaris o generar visualitzacions. Entre els seus mètodes més importants es troben init(LexModel model, LexView view, NotificationService notificationService), que configura les connexions entre els components, i handleEvent(Event event), que processa els esdeveniments notificats pel model o altres components.

El mètode onEvent(Event event) és fonamental per gestionar els esdeveniments generats pel model o altres parts de l'aplicació. Aquest mètode utilitza un mecanisme de commutació (switch) per identificar el tipus d'esdeveniment i executar l'acció corresponent. Per exemple, pot actualitzar la vista amb un resum del corpus carregat, mostrar el progrés del càlcul de la matriu de distàncies, o renderitzar visualitzacions com taules, grafos o arbres filogenètics. També gestiona errors de càlcul, mostrant missatges d'error a la vista quan sigui necessari. Això assegura una comunicació fluida i reactiva entre els components de l'aplicació.

El NotificationService és una interfície que defineix el mecanisme de publicació i subscripció d'esdeveniments dins l'aplicació, facilitant la comunicació entre els diferents components. Els seus mètodes principals són publish(Event event), que permet publicar un esdeveniment perquè altres components el rebin, i subscribe(EventType type, EventListener listener), que registra un observador per a un tipus d'esdeveniment específic.

La classe NotificationServiceImpl és la implementació concreta de NotificationService. Gestiona la distribució d'esdeveniments utilitzant una estructura interna per emmagatzemar els observadors registrats. Els seus mètodes clau inclouen publish(Event event), que notifica tots els observadors subscrits a un tipus d'esdeveniment, i unsubscribe(EventType type, EventListener listener), que elimina un observador registrat. Aquesta classe assegura una comunicació fluida i desacoblada entre els components del programa.

F. Flux d'Esdeveniments de l'Aplicació

El funcionament intern del sistema es basa en un model d'interacció orientat a esdeveniments, estructurant la comunicació entre els diferents components mitjançant un patró de control centralitzat. En l'Annex A, es presenta un diagrama de seqüència que il·lustra el flux complet d'esdeveniments durant l'execució del càlcul de distàncies de Levenshtein dins de l'aplicació.

El flux comença amb l'arrencada de l'aplicació, on l'usuari interactua amb la interfície gràfica (Vista). Aquesta instància el controlador principal i registra els canals de notificació asíncrona a través del servei de notificacions. L'usuari introdueix els paràmetres de l'execució —com la selecció d'arxius d'idiomes a emprar, o el tipus de visualització— mitjançant els elements interactius de la Vista. En prémer el botó d'execució (Compute), el controlador valida els paràmetres i llança la tasca de càlcul en segon pla per mantenir la interfície fluida.

El càlcul de les distàncies de Levenshtein (o els altres algorismes seleccionables) es realitza sobre les dades proporcionades, i les notificacions de progrés es propaguen cap a la Vista mitjançant el servei de notificacions. Això permet actualitzar la interfície en temps real, mostrant informació contextual de l'estat actual, com és la barra de progrés.

Un cop finalitzat el càlcul, el controlador sol·licita la construcció de la matriu de distàncies o l'estructura de visualització corresponent. El resultat es notifica novament a la Vista, que s'encarrega de mostrar gràficament l'arbre, ressaltar els nodes rellevants i habilitar funcionalitats com l'enfocament automàtic o l'exportació de resultats, segons les opcions seleccionades per l'usuari.

La interacció conclou amb la possibilitat d'explorar l'arbre generat, centrar la vista en nodes concrets o consultar detalls addicionals sobre les distàncies calculades. Durant tot el procés, el sistema incorpora mecanismes de gestió d'errors: si es detecta algun problema durant el càlcul o la visualització, es comunica immediatament a la Vista, que actualitza l'estat de la interfície i informa l'usuari de l'incident.

Aquest enfocament modular i basat en esdeveniments permet mantenir una separació clara de responsabilitats entre les capes de l'aplicació, assegurant una resposta àgil i fluida a les accions de l'usuari.

IV. Algorismes implementats, optimitzacions i paral·lelisme

Aquesta secció reuneix diferents explicacions sobre els algorismes implementats i les optimitzacions realitzades, juntament es descriu breument els costos computacionals i de memòria de cada algoritme. Seguidament es farà una explicació de la concurrència implementada en els algorismes i en l'estructura general del programa.

A. Algorismes implementats i optimitzacions

La detecció de semblança lèxica entre paraules es pot abordar mitjançant diverses mesures de distància o similitud. En aquesta secció s'estudien quatre algorismes clàssics per a comparar cadenes de caràcters: la distància de Levenshtein, la distància de Damerau-Levenshtein, la longitud de la subseqüència comuna més llarga (LCS, de l'anglès Longest Common Subsequence) i la similitud de Jaro-Winkler. Per a cadascun d'aquests, es presenta una descripció teòri-

ca formal, el pseudocodi de l'algorisme corresponent (en notació de *programació dinàmica* o procedimental segons convingui), les optimitzacions aplicades i una anàlisi de la complexitat temporal i espacial.

1) Distància de Levenshtein

La distància de Levenshtein, introduïda per Vladimir Levenshtein (1965), es defineix com el nombre mínim d'operacions d'edició necessàries per transformar una cadena de caràcters X en una altra cadena Y. Les operacions permeses són: inserció d'un caràcter, supressió (eliminació) d'un caràcter, i substitució d'un caràcter per un altre. Cada operació té un cost unitari en la formulació clàssica, de manera que la distància correspon al total mínim d'edits aplicats. Formalment, si denotem per $d(i,j)$ la distància de Levenshtein entre els prefixos $X[1..i]$ i $Y[1..j]$ (és a dir, entre els i primers caràcters de X i els j primers de Y), aquesta es pot calcular recursivament com:

$$d(i,j) = \begin{cases} \max(i,j), \\ \text{si } \min(i,j) = 0, \\ \min\Big\{ d(i{-}1,j)+1,\ d(i,j{-}1)+1, \\ d(i{-}1,j{-}1){+} \\ \delta(X_i,Y_j)\Big\}, \\ \text{altrament.} \end{cases}$$

on $\delta(X_i,Y_j) = \begin{cases} 0 & \text{si } X_i = Y_j, \\ 1 & \text{si } X_i \neq Y_j \end{cases}$ i s'assumeix $d(0,j) = j$ i $d(i,0) = i$ per a qualsevol i,j. En paraules, si almenys una de les dues subcadenes és buida, la distància és la llargada de l'altra (calen tantes insercions o supressions com caràcters tingui l'altra cadena); altrament, es pren el mínim cost entre eliminar un caràcter de X ($d(i-1,j)+1$), inserir un caràcter en X ($d(i,j-1)+1$), o substituir X_i per Y_j ($d(i-1,j-1)+1$ si $X_i \neq Y_j$, o bé $+0$ si coincideixen). La implementació clàssica d'aquest càlcul és l'algorisme de Wagner-Fischer, que empra una matriu de dimensió $(n+1)\times(m+1)$ (sent $n = |X|$ i $m = |Y|$) per a computar $d(n,m)$ de forma iterativa. A continuació es presenta el pseudocodi del procediment:

Aquest algorisme té una complexitat temporal $O(n \cdot m)$, ja que s'omple una taula de $(n+1)\times(m+1)$ elements, i complexitat espacial $O(n \cdot m)$ si es manté tota la matriu en memòria. No obstant això, es pot optimitzar l'ús de memòria observant que per calcular la fila i-èsima només calen la fila immediatament anterior i la fila actual; així, és possible reduir l'espai a $O(\min(n,m))$ utilitzant només dos vectors de longitud $m+1$ (o $n+1$) que es van reutilitzant. Una altra optimització és l'aplicació de llindars: si només interessa saber si la distància és menor que un cert valor k, es pot interrompre el càlcul tan aviat com es detecta que fan falta més de k operacions (per exemple, si en processar els prefixos s'obté $d(i,j) > k$ per a algun cas, o fins i tot prèviament, notant que si $|n-m| > k$ aleshores $d(n,m) > k$ automàticament). Amb aquestes millores, la distància de Levenshtein es pot calcular de manera eficient fins i tot per a cadenes relativament llargues, i és àmpliament utilitzada com a mesura bàsica de semblança entre paraules.

Algorithm 1 Càlcul de la distància de Levenshtein

Require: Cadenes X i Y de longitud n i m respectivament
Ensure: Valor $d[n][m]$, distància Levenshtein entre X i Y

```
Crear matriu d[0..n][0..m]
for i = 0 to n do
    d[i][0] ← i
end for
for j = 0 to m do
    d[0][j] ← j
end for
for i = 1 to n do
    for j = 1 to m do
        cost ← if X[i] = Y[j] then 0 else 1
        d[i][j] ← min(d[i-1][j] + 1, d[i][j-1] + 1, d[i-1][j-1] + cost)
    end for
end for
return d[n][m]
```

2) Distància de Damerau-Levenshtein

La distància de Damerau-Levenshtein amplia la distància de Levenshtein permetent, a més, l'operació de transposició de dos caràcters adjacents (intercanviar-ne la posició) com una sola edició. Aquesta variant deu el seu nom a Frederick J. Damerau, qui el 1964 va destacar la importància de considerar les transposicions juntament amb insercions, supressions i substitucions en l'anàlisi d'errors tipogràfics. Formalment, la distància de Damerau-Levenshtein entre X i Y es pot definir de manera similar a la de Levenshtein, afegint un cas a la recurrència per contemplar transposicions. El recurrerent de l'algorisme de programació dinàmica es modifica així:

$$d(i,j) = \min\{\ldots,\ d(i{-}2,j{-}2)+1\} \quad \text{si } i>1,\ j>1, \\ X_i = Y_{j-1} \text{ i } X_{i-1} = Y_j,$$

és a dir, si els dos últims caràcters dels prefixos considerats (de longitud i i j, respectivament) estan intercanviats entre X i Y, es pot corregir aquesta transposició amb una única operació (a més de les operacions estàndard d'inserció, supressió i substitució ja considerades). El pseudocodi següent mostra l'algorisme complet, que és una extensió directa de l'anterior de Wagner-Fischer:

Algorithm 2 Càlcul de la distància de Damerau-Levenshtein

Require: Cadenes X i Y de longitud n i m
Ensure: Valor $d[n][m]$
Crear matriu $d[0..n][0..m]$
for $i = 0$ to n **do**
    $d[i][0] \leftarrow i$
end for
for $j = 0$ to m **do**
    $d[0][j] \leftarrow j$
end for
for $i = 1$ to n **do**
    for $j = 1$ to m **do**
        $cost \leftarrow$ if $X[i] = Y[j]$ then 0 else 1
        $d[i][j] \leftarrow \min\big(d[i-1][j] + 1,\ d[i][j-1] + 1,\ d[i-1][j-1] + cost\big)$
        if $i > 1$ and $j > 1$ and $X[i] = Y[j-1]$ and $X[i-1] = Y[j]$ **then**
            $d[i][j] \leftarrow \min(d[i][j], d[i-2][j-2] + 1)$
        end if
    end for
end for
return $d[n][m]$

La complexitat computacional d'aquest algorisme continua sent $O(n \cdot m)$ en temps, ja que simplement afegeix una comparació constant per iteració dins el doble bucle. En espai també és $O(n \cdot m)$ en la implementació directa. De forma anàloga a Levenshtein, es podria reduir l'ús de memòria mantenint només unes poques files de la matriu a la vegada; però en aquest cas cal almenys conservar informació de la fila anterior i de la fila encara anterior (per accedir a $d[i-2, j-2]$) per tal de calcular les transposicions, fet que complica lleugerament l'optimització de l'espai. Malgrat aquesta petita sobrecàrrega, l'algorisme de Damerau-Levenshtein és manejable per a longituds de paraules moderades i resulta especialment útil quan es volen detectar intercanvis de lletres de forma eficient (per exemple, en correcció ortogràfica o en comparació de noms propis, on errors de teclat per invertir dues lletres són relativament comuns).

3) Subseqüència Comuna Més Llarga (LCS)

La *subseqüència comuna més llarga* entre dues cadenes X i Y és la seqüència de caràcters més llarga que es pot obtenir de totes dues cadenes mantenint-ne l'ordre (no cal que els caràcters siguin consecutius, però sí respectant la posició relativa). La longitud d'aquesta subseqüència (denotada LCS, de l'anglès Longest Common Subsequence) proporciona una mesura de similitud entre X i Y: com més gran és la LCS, més caràcters tenen en comú en ordre relatiu. En termes de distància d'edició, si només es permeten operacions d'inserció i supressió (sense substitucions), la distància mínima per transformar X en Y ve donada per $d_{ins\text{-}del}(X, Y) = n + m - 2 \cdot LCS(X, Y)$. Aquesta expressió s'interpreta fàcilment: per convertir X en Y només amb insercions i eliminacions, cal eliminar els caràcters que no formen part de la subseqüència comuna i inserir en les posicions corresponents els caràcters restants de Y (en total, $|X| - LCS$ eliminacions i $|Y| - LCS$ insercions). El càlcul de la LCS es pot dur a terme amb un algorisme de programació dinàmica, anàleg al de Levenshtein però maximitzant coincidències en lloc de minimitzar edicions. Es defineix $L(i, j)$ com la longitud de la subseqüència comuna més llarga entre els prefixos $X[1..i]$ i $Y[1..j]$. Llavors:

$$L(i,j) = \begin{cases} 0, & \text{si } i = 0 \text{ o } j = 0, \\ L(i-1, j-1) + 1, & \text{si } X_i = Y_j, \\ \max\{L(i-1, j),\ L(i, j-1)\}, & \text{si } X_i \neq Y_j. \end{cases}$$

amb condicions inicials $L(i, 0) = 0$ i $L(0, j) = 0$. Aquesta recurrència omple una taula $L[0..n,\ 0..m]$ on cada entrada $L(i, j)$ conté la longitud de la LCS dels prefixos corresponents, i en particular $L(n, m)$ resulta la longitud de la LCS de X i Y. El pseudocodi següent implementa aquest algorisme pas a pas:

Algorithm 3 Càlcul de la longitud de la LCS

Require: Cadenes X i Y de longitud n i m
Ensure: Valor $L[n][m]$, longitud de la subseqüència comuna
Crear matriu $L[0..n][0..m]$ inicialitzada a 0
for $i = 1$ to n **do**
    for $j = 1$ to m **do**
        if $X[i] = Y[j]$ **then**
            $L[i][j] \leftarrow L[i-1][j-1] + 1$
        else
            $L[i][j] \leftarrow \max(L[i-1][j], L[i][j-1])$
        end if
    end for
end for
return $L[n][m]$

La complexitat d'aquest algorisme és $O(n \cdot m)$ en temps, ja que cal considerar totes les parelles de posicions (i, j) per omplir la taula. La complexitat espacial és també $O(n \cdot m)$ si es guarda la taula completa L (necessària si es vol reconstruir explícitament una subseqüència comuna òptima). No obstant, si només interessa calcular la longitud de la LCS (sense recuperar la subseqüència en si), es pot reduir l'espai a $O(\min(n, m))$ utilitzant el mateix truc de guardar només la fila anterior i la fila actual en cada iteració. A més, existeix l'algorisme de Hirschberg que permet recuperar una LCS completa utilitzant només $O(n + m)$ espai addicional, dividint la feina de la taula en subproblemes més petits (tot mantenint el temps $O(n \cdot m)$). En la pràctica, l'algorisme de la LCS és eficient per a longituds de paraules relativament curtes i resulta útil per mesurar la similitud "estructural"entre paraules (quant de substrat en comú tenen), complementant la informació proporcionada per les distàncies d'edició.

4) **Similitud de Jaro-Winkler**

La *similitud de Jaro-Winkler* és un mètode que calcula la semblança entre dues cadenes en un rang normalitzat de 0 a 1 (on 1 indica que les cadenes són idèntiques). Es tracta d'una versió millorada de la similitud de Jaro, amb un ajust introduït per William E. Winkler el 1990 per a tenir en compte els prefixos coincidents. A diferència de les mesures basades en edicions, la mètrica de Jaro-Winkler es fonamenta en comparar els caràcters compartits i la seva posició: primer es determinen quins caràcters de X tenen correspondència en Y (i viceversa) dins d'un cert abast posicional, i després es quantifica quants d'aquests caràcters coincidents es troben desordenats o allunyats en les dues cadenes. En la definició formal, sigui M el nombre de caràcters *matching* (coincidents) entre X i Y. Un caràcter X_i es considera que fa *match* amb un caràcter Y_j si $X_i = Y_j$ i la diferència de posició entre i i j és com a màxim $\left\lfloor \frac{\max(n,m)}{2} \right\rfloor - 1$ (on $n = |X|$ i $m = |Y|$), de manera que només es compten correspondències de caràcters relativament a prop en longitud. També es defineix T com la meitat del nombre de caràcters matching que no estan en el mateix ordre en les dues cadenes (és a dir, el nombre de transposicions). Llavors, la similitud de Jaro sim_J ve donada per:

$$\mathrm{sim}_J(X,Y) = \begin{cases} 0, & \text{si } M = 0, \\ \frac{1}{3}\left(\frac{M}{n} + \frac{M}{m} + \frac{M-T}{M}\right), & \text{altrament,} \end{cases}$$

on M és el nombre de matchings i T és el nombre de transposicions (ja dividit per 2). Aquesta similitud és 0 si no hi ha cap caràcter en comú, i 1 si les dues cadenes són exactament iguals (en aquest cas $M = n = m$ i $T = 0$ perquè no hi ha cap caràcter desordenat). Winkler va proposar ajustar aquesta mesura per tal de donar més pes als prefixos coincidents. Empíricament, en comparar paraules (per exemple, noms propis) és habitual considerar que les coincidències al començament de la paraula tenen més rellevància que les del final. Sigui ℓ la longitud del prefix comú inicial de X i Y (per exemple, $\ell = 0$ si comencen diferent, $\ell = 3$ si comparteixen els tres primers caràcters, etc., amb un límit superior usual de $\ell_{\max} = 4$). Definint un factor d'ajust p (per exemple $p = 0.1$), la similitud de Jaro-Winkler sim_{JW} es calcula com:

$$\mathrm{sim}_{JW}(X,Y) = \mathrm{sim}_J(X,Y) + \ell \cdot p \cdot (1 - \mathrm{sim}_J(X,Y)),$$

on ℓ es limita a $\ell_{\max}$ en cas que el prefix comú sigui molt llarg. Així, sim_{JW} també oscil·la entre 0 (cap semblança) i 1 (identitat completa), i atorga un increment addicional a sim_J proporcional a la longitud del prefix inicial compartit. A continuació es descriu l'algorisme pas a pas per calcular sim_{JW}. En essència, es realitzen tres passos: (1) trobar els caràcters matching, (2) comptar transposicions, i (3) aplicar la fórmula de Jaro-Winkler amb l'ajust de prefix:

Algorithm 4 Càlcul de la similitud de Jaro-Winkler

Require: Cadenes X i Y de longitud n, m, paràmetres p i $\ell_{\max}$

Ensure: Valor $sim_{JW}(X,Y)$

1: Calcular coincidències M dins la finestra $\lfloor \frac{\max(n,m)}{2} \rfloor - 1$
2: Comptar transposicions T (caràcters coincidents en ordre incorrecte)
3: **if** $M = 0$ **then return** 0
4: **end if**
5: $sim_J \leftarrow \frac{1}{3}\left(\frac{M}{n} + \frac{M}{m} + \frac{M-T}{M}\right)$
6: $L \leftarrow$ longitud del prefix comú (fins $\ell_{\max}$)
7: $sim_{JW} \leftarrow sim_J + L \cdot p \cdot (1 - sim_J)$
8: **return** sim_{JW}

L'anàlisi de costos d'aquest algorisme mostra un pitjor cas de $O(n \cdot m)$ comparacions, atès que en el doble bucle inicial cada caràcter de X pot arribar a comparar-se amb diversos caràcters de Y dins del rang delimitat per $maxDist$. En la pràctica, però, la finestra limitada redueix significativament el nombre de comparacions quan les cadenes són llargues i només comparteixen pocs caràcters. L'ús de memòria és $O(n + m)$ per emmagatzemar els arrays booleans i algunes variables auxiliars, la qual cosa és molt assumible en comparar paraules de llargada moderada. Cal destacar que en implementacions habituals s'estableix un llindar (per exemple $sim_J > 0.7$) per decidir si s'aplica l'ajust de prefix de Winkler, de manera que només s'incrementa la similitud quan ja hi ha una semblança base important (això evita afavorir cadenes amb prefixos iguals però similitud global baixa). La mesura de Jaro-Winkler és especialment efectiva en la comparació de termes curts (p. ex. noms propis o paraules similars) on petites permutacions o errors de teclat són probables: proporciona un índex de similitud acotat i interpretable directament. En canvi, per a cadenes més llargues o amb diferències distribuïdes, el valor de sim_{JW} pot no reflectir de manera lineal el nombre d'ediscrepàncies, i en aquests casos pot ser més informativa una distància d'edició absoluta. En resum, cada mètode presenta punts forts i limitacions en el context de la comparació lèxica entre idiomes. La distància de Levenshtein proporciona una mesura directa i interpretativa del nombre de canvis entre dues paraules, funcionant especialment bé quan les diferències són menors (per exemple, variacions ortogràfiques o afegir/eliminar un accent o una lletra muda). La variant de Damerau- Levenshtein és més adequada quan hi ha transposicions de lletres, ja que evita penalitzar-les doblement i reflecteix millor la semblança percebuda en casos on dues lletres estan intercanviades. Per la seva banda, la LCS captura la idea de substrat comú: identifica si ambdues paraules comparteixen una seqüència significativa de lletres en el mateix ordre, la qual cosa és útil per detectar cognats amb arrels comunes fins i tot si difereixen en prefixos o sufixos. No obstant això, la LCS no distingeix entre substitucions i insercions (tota diferència és un caràcter que no fa match),

de manera que pot sobreestimar la similitud quan l'ordre de les lletres canvia o hi ha lletres diferents enmig. Finalment, la similitud de Jaro-Winkler ofereix un percentatge de semblança acotat i fàcil d'interpretar, sent molt eficaç per a paraules curtes on un valor alt indica clarament que les dues formes són gairebé idèntiques. Jaro-Winkler aprofita el pes del prefix comú i tolera lleugeres permutacions de caràcters, fet que la fa indicada per reconèixer variants tipogràfiques o ortogràfiques. Tanmateix, pot resultar menys precisa en paraules molt llargues o quan la similitud és intermèdia, ja que un valor proporcional pot no correspondre exactament al nombre d'edicions. En conclusió, en un projecte de detecció de semblança lèxica convé escollir (o combinar) la mètrica segons el tipus de relació entre les paraules: la distància de Levenshtein (o Damerau) per comptabilitzar diferències puntuals lletra a lletra, la LCS per descobrir trams comuns significatius, i Jaro-Winkler per obtenir un índex normalitzat de similitud que ressalti les coincidències prefixals i penalitzi suaument les permutacions adjacents.

B. Gestió de concurrència i paral·lelisme

La implementació del projecte **LexDistance** aprofita el paral·lelisme de maquinari per accelerar el càlcul intensiu de distàncies lèxiques entre idiomes, alhora que manté la interfície gràfica responsiva. El **model de dades** (classe `LexModel`) incorpora un component de concurrència basat en un *pool* de fils (`ExecutorService`) configurat segons els processadors disponibles del sistema. En construir el model, es crea un *thread pool* amb una mida adaptable: s'estableix un nombre de fils bàsics equivalent, com a mínim, a 2 o a la meitat dels nuclis CPU disponibles, i un màxim que arriba als nuclis totals (mínim 4 fils). D'aquesta manera, el sistema pot llançar múltiples fils de treball en paral·lel per dur a terme càlculs costosos, utilitzant eficientment els recursos *multicore*. Les tasques pesades (com ara el càlcul de la matriu de distàncies) s'executen de forma asíncrona en aquest *pool* de fils, evitant bloquejar el fil d'interfície gràfica durant el procés.

1) Divisió de subtasques per al càlcul de distàncies

El càlcul de la **matriu de distàncies** entre idiomes s'ha dissenyat seguint un patró *fork/join* per explotar el paral·lelisme intern de l'algorisme. Quan l'usuari inicia la comparació, el model (`LexModel`) encapsula l'operació en una tasca concurrent i la passa al *pool* de fils mitjançant `executorService.submit(...)`. Aquesta crida retorna immediatament un objecte `Future`, iniciant l'execució en segon pla mentre la interfície continua operativa.

Dins del càlcul, la matriu es construeix en paral·lel dividint el treball en subtasques recursives. En concret, s'utilitza un esquema recursiu de tipus *Fork/Join*, on una classe interna representa la tasca de calcular distàncies per a un subconjunt de files. El fil inicial crea una tasca principal que cobreix tota la matriu i la submeteix a un `ForkJoinPool` dedicat, amb un grau de paral·lelisme proper a `Runtime.getRuntime().availableProcessors() - 1`.

Cada tasca comprova la mida del seu bloc de treball i decideix si ha de dividir-se o executar-se directament. **S'aplica un llindar dinàmic** que determina el mode seqüencial versus paral·lel: si la mida de la tasca és inferior al llindar, es calcula directament; si no, es divideix. Aquest llindar (entre 1 i 5 files habitualment) evita sobrecàrrega en tasques petites i millora el rendiment global.

Les subtasques recorren blocs de files i per cada fila la comparen amb les columnes superiors (matriu triangular superior), emmagatzemant els resultats simètricament. L'accés concurrent és segur perquè cada fil tracta blocs disjunts. Paral·lelament, un `AtomicInteger` compta les comparacions completades per monitoritzar el progrés de manera segura. **La sincronització final** es fa mitjançant `join()`, garantint que totes les subtasques han acabat abans de continuar. Quan la matriu és completa, el model emet un esdeveniment indicant que el càlcul ha finalitzat.

2) Comunicació asíncrona amb la interfície d'usuari

La interfície gràfica, desenvolupada amb JavaFX, manté una alta responsivitat gràcies a una arquitectura de notificacions desacoblada. Quan el model finalitza una tasca (com carregar diccionaris o calcular distàncies), no interactua directament amb la vista, sinó que **publica esdeveniments** a través d'un `NotificationService`. El controlador de la UI, registrat com a *listener*, rep aquests esdeveniments a través de la implementació `NotificationServiceImpl`.

Si l'esdeveniment afecta la interfície, el servei utilitza `Platform.runLater(...)` per transferir l'actualització al fil principal de JavaFX de manera segura. Això garanteix que cap modificació visual (com actualitzar una barra de progrés o mostrar un gràfic) interfereixi amb l'execució dels càlculs paral·lels.

Aquest enfocament permet actualitzacions visuals en temps real sense bloquejos. Per exemple, mentre es calcula la matriu de distàncies, la barra de progrés s'actualitza asíncronament, i quan el càlcul acaba, la vista es refresca per mostrar els resultats. Gràcies a aquesta arquitectura basada en `ExecutorService`, `Future`, i `Platform.runLater`, es manté una separació clara entre càlcul i visualització, permetent un funcionament fluït de l'aplicació fins i tot amb càrregues intensives de treball.

V. Conclusió

Els algorismes implementats per calcular distàncies (Levenshtein, Damerau-Levenshtein, Jaro-Winkler i LCS, algoritme de la subseqüència comuna més llarga) ofereixen solucions diverses amb característiques diferenciades. El Levenshtein és un mètode senzill i ben conegut, versàtil per comparar diferents tipus de seqüències; tanmateix, presenta una complexitat computacional elevada (d'ordre quadràtic),

la qual cosa pot fer-lo poc eficient amb seqüències de longitud elevada. El Damerau-Levenshtein estén el model incorporant la gestió de transposicions, fet que millora la precisió davant errors comuns d'inversió de caràcters, tot i que complica l'estructura i afegeix sobrecàrrega de càlcul. Jaro-Winkler, especialitzat en la comparació de cadenes curtes (per exemple, noms propis), sol ser més eficient en aquests escenaris i premia les coincidències de prefixos; però la seva aplicabilitat es limita a cadenes breus i no està dissenyat per a gestionar variants d'error més complexes, com ho fan Levenshtein i Damerau-Levenshtein. Finalment, l'algorisme LCS mesura la similitud a través de la longitud de la subseqüència comuna més llarga; aquest enfocament proporciona informació valuosa sobre l'alineament global de seqüències, però no constitueix una distància simètrica tradicional i pateix una elevada complexitat de càlcul. En conjunt, cada mètode té punts forts i febles: Levenshtein i Damerau-Levenshtein són versàtils i adients per a diversos contextos de seqüències, a costa d'un elevat cost de processament; Jaro-Winkler és eficaç per a cadenes curtes amb coincidències de prefixos, a canvi d'una aplicabilitat limitada; i LCS aporta una perspectiva complementària de similitud seqüencial, tot i requerir més recursos computacionals. Pel que fa al rendiment global de l'aplicació, cal destacar que l'eina desenvolupada ha gestionat les distàncies amb una eficiència raonable per a conjunts de dades de mida moderada, la qual cosa resulta satisfactòria en el context del projecte. L'arquitectura Model-Vista-Controlador ha facilitat una separació clara de responsabilitats: el Model conté la lògica de càlcul i la gestió de dades, la Vista s'encarrega de la presentació i interacció amb l'usuari, i el Control actua com a intermediari en el flux d'informació entre ambdues. Aquesta modularitat ha permès organitzar el codi de manera més neta i ha simplificat l'extensió de funcionalitats (per exemple, l'addició de nous algoritmes o la modificació de la interfície) amb un impacte mínim en la resta del sistema. Pel que fa al rendiment operatiu, l'estructura MVC introdueix certa sobrecàrrega (deguda a la coordinació entre capes i la generació d'elements d'interfície), però aquest cost es veu compensat pels avantatges en mantenibilitat i escalabilitat. En termes generals, l'aplicació ha demostrat ser robusta i estable dins dels paràmetres previstos, tot i que hi ha marge de millora en entorns amb volums de dades molt alts o requisits de processament en temps real. Finalment, es poden proposar diverses línies de millora tècnica i metodològica. En primer lloc, caldria optimitzar els algorismes actuals o explorar alternatives més eficients: per exemple, mitjançant execució concurrent (ús de múltiples fils d'execució o GPU) o incorporant heurístiques que redueixin la complexitat computacional. A més, convindria ampliar la varietat de mètodes de comparació per incloure noves tècniques (com ara distàncies geomètriques, aprenentatge automàtic o índexs espacials), cosa que enriquiria l'abast de l'eina. Pel que fa a l'arquitectura, seria recomanable integrar proves automatitzades (unitàries) i sistemes de monitorització de rendiment per augmentar la fiabilitat del codi, així com enfortir la interfície d'usuari amb visualitzacions més avançades o elements interactius que facilitin l'explotació dels resultats. Aquestes millores contribuirien sens dubte a enfortir la robustesa, el rendiment i la versatilitat del projecte en futurs desenvolupaments.

Referències

[1] Oracle, *Fork/Join Framework*, Java Tutorials, 2014.
[2] S. Klymenko, *Java ForkJoinPool: A Comprehensive Guide*, Medium, 2021.
[3] A. Dix et al., *Model-View-Controller (MVC) and Observer*, Univ. of North Carolina, 2016.
[4] Sommerville, I., *Software Engineering*, 10th ed., Addison-Wesley, 2015.
[5] Chacon, S. and Straub, B., *Pro Git*, Apress, 2014.
[6] Cockburn, A., *Agile Software Development: The Cooperative Game*, Addison-Wesley, 2006.
[7] Loeliger, J. and McCullough, M., *Version Control with Git*, O'Reilly Media, 2012.
[8] Sutherland, J., *Scrum: The Art of Doing Twice the Work in Half the Time*, Crown Business, 2014.
[9] Gamma, E., Helm, R., Johnson, R. and Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.
[10] Reenskaug, T., *Models, Views and Controllers*, 1979.
[11] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M., *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*, Wiley, 1996.
[12] Dix, A., Finlay, J., Abowd, G. and Beale, R., *Human-Computer Interaction*, Prentice Hall, 2004.
[13] Fowler, M., *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2003.
[14] Joshua Bloch, *Creating and Destroying Java Objects*, Drdobbs, 2008
[15] Goetz, B. et al., *Java Concurrency in Practice*, Addison-Wesley, 2006.
[16] Lea, D., *Concurrent Programming in Java: Design Principles and Patterns*, Addison-Wesley, 2000.
[17] Amdahl, G. M., *Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities*, AFIPS Conference Proceedings, 1967.
[18] Herlihy, M. and Shavit, N., *The Art of Multiprocessor Programming*, Morgan Kaufmann, 2008.
[19] Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C., *Introduction to Algorithms*, 3rd ed., MIT Press, 2009.
[20] Levenshtein, V. I., *Binary codes capable of correcting deletions, insertions, and reversals*, Soviet Physics Doklady, Vol. 10, No. 8, pp. 707-710, 1966.
[21] Wagner, R. A. and Fischer, M. J., *The String-to-String Correction Problem*, Journal of the ACM, Vol. 21, No. 1, pp. 168-173, 1974.
[22] Damerau, F. J., *A technique for computer detection and correction of spelling errors*, Communications of the ACM, Vol. 7, No. 3, pp. 171-176, 1964.
[23] Hirschberg, D. S., *A linear space algorithm for computing maximal common subsequences*, Communications of the ACM, Vol. 18, No. 6, pp. 341-343, 1975.
[24] Jaro, M. A., *Advances in record-linkage methodology as applied to matching the 1985 census of Tampa, Florida*, Journal of the American Statistical Association, Vol. 84, No. 406, pp. 414-420, 1989.
[25] Winkler, W. E., *String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage*, Proceedings of the Section on Survey Research Methods, American Statistical Association, pp. 354-359, 1990.
[26] Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C., *Introduction to Algorithms*, 3rd ed., MIT Press, 2009.
[27] Kleinberg, J. and Tardos, É., *Algorithm Design*, Pearson, 2006.
[28] Aho, A. V., Hopcroft, J. E. and Ullman, J. D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.
[29] Sipser, M., *Introduction to the Theory of Computation*, 3rd ed., Cengage Learning, 2012.
[30] Tarjan, R. E., *Data Structures and Network Algorithms*, SIAM, 1983.

Annex:

A. Diagrama de fluxe

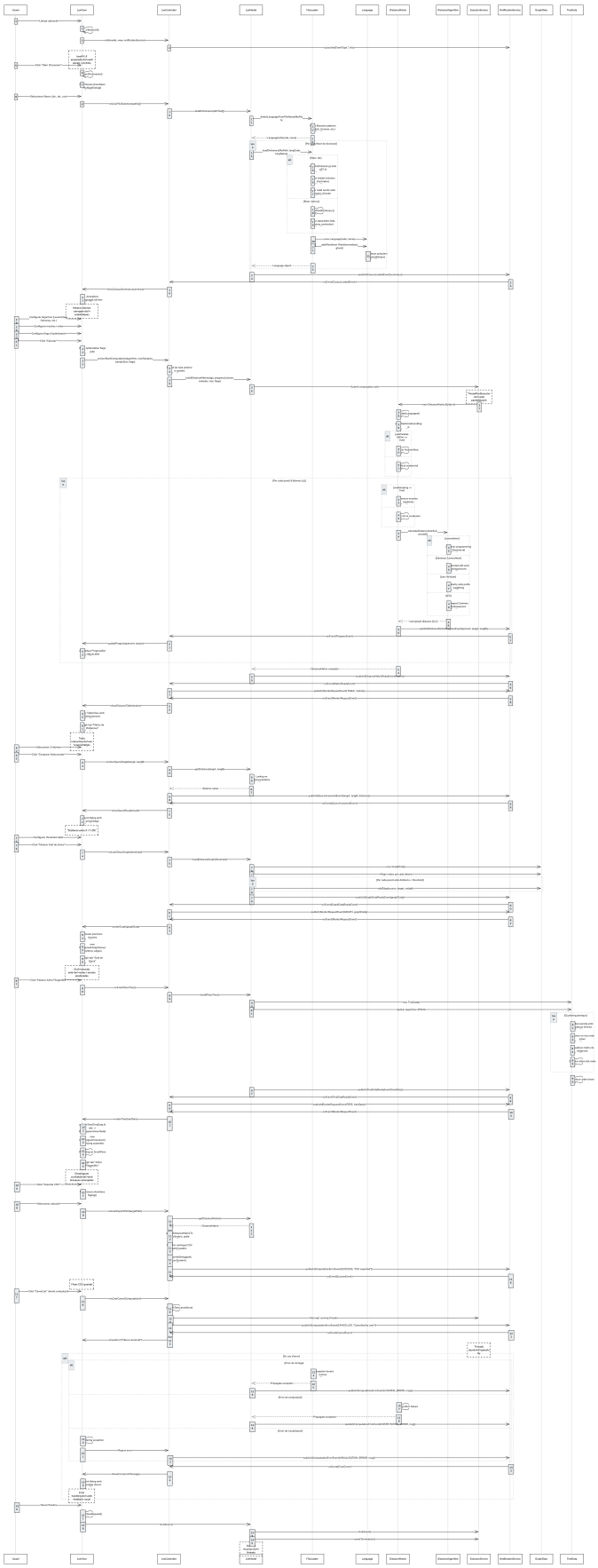

Figura 1. **Diagrama de fluxe**