Aspecte ale lingvisticii computaționale în producerea de cuvinte

Lucrare de Licență

Maria-Smaranda Pandele

Coordonator: Prof. Dr. Liviu Dinu

Facultatea de Matematică și Informatică, UNIBUC

Rezumat

In aceasta lucrare ne propunem sa introducem cateva notiuni despre lingvistica computationala si producerea de cuvinte. Pornind de la tehnici deja existente, examinam o metoda prin care sa putem produce cuvinte cognate ce lipsesc din anumite limbi romanice, cu precadere pe limba romana.

Cuprins

Capitolul 1

Introducere

Limbajul unei tari se afla in continua schimbare datorita mai multor cauze precum cauze sociale, economice, migratia populatiei, progresele in stiinta, technologie si medicina. De-a lungul timpului, modificarile s-au petrecut inevitabil si nu au fost neaparat reglementate de experti in domeniu. Astfel a aparut o noua ramura a lingvisticii ce studiaza sistematic aceste transformari incercand sa gaseasca sabloane, reguli si sa puna o ordine asupra schimbarilor lexicale, fonetice, semantice si sintactice. Cateva exemple clasice ar fi determinarea etimologiei unui cuvant (romanescul *genunchi* provine din latinescul *genuclum*) sau determinarea similaritatii limbajelor.

De cele mai multe ori, popoare ce vorbeau aceeasi limba s-au despartit si au evoluat diferit, aparand limbaje derivate. Ne putem da seama de gradul de rudenie a acestora prin identificarea formelor **cognate** (grupuri de cuvinte ce au derivat din acelasi etimon). In alte cazuri, limbiile au imprumutat cuvinte intre ele fie luandu-le ca atare (cuvantul japonez *sushi*), fie modificand forma lor (romanescul *cafea* din turcescul *kahve*). Astfel, se pot descoperi chiar relatii de natura istorica intre limbi si popoarele ce le folosesc. Spre exemplu, exista peste 200 de cuvinte ce se regasesc in toate limbile romanice moderne mai putin limba romana. Este greu de crezut ca o parte din aceste cuvinte nu au existat in lexicul limbii romane la un moment dat. Fischer[7] a identificat cateva cauze ce au condus la disparitia acestor cuvinte si inlocuirea acesora prin cuvinte cu alte etimologi: cauze externe, social-econimice, schimbarea ocupatiilor romanilor, intreruperea Romaniei cu lumea Occidentala, dezvoltarea limbii romane departe de nucelul romanic si asa mai departe.

1.1 Reconstructia limbajelor folosind metode comparative

Lingvisticii istorici se ocupa cu cercetarea schimbarilor limbilor de-a lungul timpului. O preocupare majora a acestora o reprezinta producerea de cuvinte inrudite. Ei folosesc metode comparative prin care analizeaza modificarile limbajelor efectuand comparatii intre limibi inrudite pentru a deduce, printr-o inginerie inversa, propietati ale limbii stramos comune.[20] Practic, se incearca determinarea grupurilor de cuvinte cognate si gasirea unor reguli prin care au fost obtinute din limba stramos. Acestea nu sunt neaparat atestate, cele mai multe dintre ele denumindu-se **proto-limbaje**. In consecinta, rezultatele obtinute sunt foarte greu de demonstrat pentru ca nu exista dovezi arheologice concrete.

Asemenea metode au dat rezolvate satisfacatoare, reusind sa determine structura famililor de limbi europene. Mai mult, reconstructia proto-limbajul **Indo-European** considerat cel mai vechi limbaj cunoscut, a fost posibila prin punerea in corelatie cu limbajele derivate din acesta (proto-limbajul German, proto-limbajul Indo-Iranian).[19] In plus, metodele comparative au avut avut succes si in analiza altor familii de limbaje de pe alte continente.

1.2 Metode computationale

In principal, tehnicile comparative au mai multi pasi efectuati manual de lingvistici si presupun multe ore de munca. In perioada in care calculatoarele nu erau inca inventate, cautarea cuvintelor in carti, dictionare si prelucrarea acestora era o munca nu numai obositoare si repetitiva dar si dificila, fiind nevoie de atentie continua. Insa, odata cu aparitia metodelor computationale, se pune accentul pe determinarea automata a cuvintelor inrudite precum in [9] sau [15]. Aceste solutii sunt departe de a inlocui un expert in domeniul lingvisticii si isi propun mai mult sa vina in ajutorul acestuia pentru a facilita dezvoltarea in profunzime a domeniului.

Metodele computationale doar automatizeaza procesul pastrand ideea pornirii de la formele moderne ale cuvintelor pentru a le reconstrui pe cele vechi din care provin. Exista numeroase date intretinute activ de catre specialisti ce determina conexiuni intre cuvinte din mai multe limbi moderne. Spre exemplu, corpusul Europarl pentru limbile oficiale vorbite in Uniunea Europeana sau WordNet (Fellbaum, 1998) pentru limba engleza. Asemenea resurse vaste sunt esentiale in aplicarea unor tehnici comparative automate. Pe baza lor se pot determina si aplica reguli de productie a cuvintelor intr-un mod formal, fara prea mari interventii din partea lingvisticilor pentru asa zisele *exceptii*.

Este dificil de prezis cum anume a fost modificat un cuvant pentru a ajunge in forma lui actuala. Desi se presupune ca ar exista sabloane si reguli in evolutia unui cuvant din etimonul sau, sunt si exemple care s-au detasat semnificativ de stramosul lor: latinescul *umbilicu(lu)s* a ajuns la foma de *buric* in romana, *nombril* in Franceza si *umbigo* in Portugheza.

Detectia automata a cuvintelor cognate poate fi formulata ca o problema de clasificare prin invatare autoamata. O serie de atribute s-au dovedid a fi de succes de-a lungul timpului precum *n*-gramele, distante de editare, cel mai lung subsir comun etc. Jager si Sofroniev [8] determina perechi cognate folosind un clasificator SVM iar apoi probabilitati si distante pentru a grupa cuvinte in grupuri cognate. Rama [14] aplica o retea neurala convolutionala. Ciobanu si Dinu [2] folosesc campuri aleatoare conditionate reusind sa reconstruiasca proto-cuvinte din seturi cognate incomplete.

1.3 Latina si limbi romanice moderne

Romana(ro), Italiana(it), Franceza(fr), Spaniola(es), Portugheza(pt) sunt doar cateva dintre limbile moderne ce au evoluat din latina, in special din dialectul vulgar (latina vorbita de oamenii de rand in Imperiul Roman). Bineinteles, provenind din aceeasi limba, se pot pune foarte multe probleme de natura lingvistica precum gradul de similaritate dintre acestea, gasirea grupurilor de cuvinte cognate, reconstructia de cuvinte latinesti neatestate si asa mai departe.

Tranzitia de la latina la o limba romanica moderna s-a efectuat prin schimbari majore de vocabular, sintaxa si fonologie. Aceste schimbari sunt mai mult sau mai putin similare pentru fiecare limba romanica moderna. In principal, s-a urmarit simplificarea vocabulurului prin eliminarea arhaismelor si a exceptilor in favoarea regulilor clare, reducerea sinonimelor, stabilirea clara a sensurilor cuvintelor (conform [18]).

1.4 Obiective si abordare

Pentru limbile romanice *ro, it, fr, es, pt* exita un dictionar etimologic in [17] pornind de a limba latina. Acest set este incomplet si ne propunem sa-l completam folosind o metoda computationala bazata pe lingvistica comparativa. Evaluarea rezultatelor va fi facuta in mare parte manual, aceste cuvinte nefiind atestate. Metoda computationala are la baza ideea prezentata in [2].

Astfel, pornind de la mai multe perechi cognate din limbi romanice moderne, se va aplica o metoda de reconstructie a cuvintelor bazata pe etichetarea secventelor si campuri aleatoare conditionate. Fiecare limba moderna va fi analizata separat in raport cu limba latina. Apoi, rezultatele din toate limbile romanice vor fi combinate folosind agregari pe baza distantei rank [4]. Toate acestea vor fi detaliate pe larg in urmatoarele capitole.

Capitolul 2

Reconstrucție de cuvinte latinesti

Pornim de la formele cuvintelor din limbile romanice moderne. Fiind date mai multe perechi de cuvinte cognate vrem sa deducem forma latinescului stramos comun. Aplicam o metoda similara cu cea folosita in [2]. Ca si in [2], ne bazam pe faptul ca modificarile ortografice sunt strans legate de evolutia cuvintelor. Deci incercam sa reconstruim proto-cuvinte din forma ortografica a cuvintelor moderne.

In final, dorim sa obtinem o lista cu n cele mai bune predictii pe care mai apoi sa le prelucam intr-o maniera atat automata cat si manuala pentru a obtine cele mai bune rezultate.

2.1 Prezentare generala

Dat fiind mai multe seturi de date de cuvinte cognate din limbi romanice moderne, metoda de reconstructie va incerca sa aproximeze forma latinescului de provenienta. Vom folosi: *Romana*, *Italiana*, *Franceza*, *Spaniola*, *Portugheza*, iar seturile de date vor avea forma $(cuvant\ modern, cuvant\ latinesc)$. Din acestea, modelul va invata pe baza campurilor conditionate aleatoare (*conditional random fields* sau prescurtat CRF) diverse schimbari de ortografie suferite de cuvintele latinesti pentru a le forma pe cele moderne. Apoi, vom aplica o tehnica de agregare a acestor rezultate pentru a combina informatie din toate limbile. In final, vom folosi mai apoi aceste schimbari pentru a oferi variante de cuvinte ce completeaza anumite seturi de date.

Pasii algoritmului pentru o anumita limba romanica moderna sunt:

1. Pentru fiecare pereche $(cuvant\ modern, cuvant\ latinesc)$, vom alinia cele doua cuvinte pentru a intelege ce semne ortografice s-au pastrat, schimbat sau elidat.

2. Pregatim antrenarea sistemului CRF: extragem caracteristici din alinierile fiecarei perechi.
3. Rulam sistemul CRF si obtinem liste de n cele mai bune productii sortate in functie de probabilitatea lor.

2.2 Aliniere

Avem perechi de tipul ($cuvant\ modern, cuvant\ latinesc$) pe care vrem sa le aliniem. Nu orice aliniere ne ofera informatie valida. Avem nevoie de asa numitele alinieri optime, in care numarul de diferente dintre cele doua cuvinte este minim. Vom aplica algoritmul de aliniere Needleman-Wunsch[13] din bioinfomatica, folosit cu succes si in probleme de procesare al limbajului natural.

2.2.1 Needleman-Wunsch

Algoritmul de aliniere Needleman-Wunsch provine din bioinfomatica, mai exact din alinierea secventelor de proteine sau nucleotide. Determinarea alinierilor se face printr-o tehnica de programare dinamica. Asadar, problema initiala va fi impartita in subprobleme fie deja calculate, fie mai usor de calculat. De fiecare cand vom spune alinieri ne vom referi doar la alinieri de tip Needleman-Wunsch.

Avem doua siruri $a = a_1a_2...a_n$ si $b = b_1b_2...b_m$ de caractere de lungime n respectiv m. Vrem sa aliniem sirul b pentru a se potrivi cu sirul a.

Exista 3 tipuri de operatii intr-o aliniere la o anumita pozitie i:

1. **Potrivire**, caracterele aliniate se potrivesc: $a_i = b_i$
2. **Nepotrivire**, caracterele aliniate nu se potrivesc: $a_i \neq b_i$
3. **Spatiu**, caracterul din primul sir nu se aliniaza cu niciun caracter din al doilea sir sau invers;

Se observa ca in cazul operatiei de tipul 3, caracterele din dreapta pozitiei curente i se vor deplasa cu o pozitie.

Fiecare dintre operatiile de mai sus are un anumit cost. In problema noastra de aliniere a unui cuvant latinesc cu un cuvant modern, vom considera costul unei operatii de **Potrvire** ca fiind 0, iar costul unei operatii de **Nepotrivire** sau **Spatiu** ca fiind 1. Atfel, alinierea optima va fi cea cu costul minim. In procesul de potrivire nu vom lua in considerare diacriticele. Literele ce contin astfel de semne vor fi considerate la fel cu literele de baza (spre exemplu, caracterul *è* poate fi potrivit cu *e*).

Problema se rezolva folosind tehnica programarii dinamice. Definim doua matrici bidimensionale cu $n+1$ si $m+1$ coloane numerotate de 0 la n respectiv de la 0 la m:

$$D_{i,j} = \text{costul minim pentru a alinia prefixul } a_1a_2...a_i \text{ din sirul a}$$
$$\text{cu prefixul } b_1b_2...b_j \text{ din sirul b}$$
$$P_{i,j} = \text{ultima operatie efectuata } \textbf{Potrivire}\text{ ,}\textbf{Nepotrivire} \text{ sau } \textbf{Spatiu}$$

Consideram faptul ca prefixul vid va fi reprezentat de pozitia fie cu linia 0 (in cazul in care prefixul vid provine din sirul a), fie cu coloana 0 (in cazul in care prefixul vid provine din sirul b).

Relatia de recurenta este:

$$D_{0,j} = j, \forall j = 0, ..., m$$
$$D_{i,0} = i, \forall i = 0, ..., n$$
$$D_{i,j} = \min \begin{cases} D_{i-1,j-1} & , a_i = b_j \textbf{ Potrivire} \\ D_{i-1,j-1} + 1 & , a_i \neq b_j \textbf{ Nepotrivire} \\ D_{i-1,j} & , \textbf{ Spatiu} \\ D_{i,j-1} & , \textbf{ Spatiu} \end{cases}$$

Scorul alinierii se va afla pe pozitia $D_{n,m}$. Pentru a reconstitui alinierea, la fiecare pas din recurenta trebuie sa retinem in matricea P matrice ultima operatie efectuata. Acum, pornim cu doi indici $i = n$ si $j = m$. Daca pentru a rezolva subproblema determinata de prefixele $a_1a_2...a_i$ si $b_1b_2...b_j$ am folosit ca si ultima operatie:

- **Potrivire**: aliniem caracterul de pe pozitia i din a cu cel de pe pozitia j din b; decrementam indicii i si j
- **Spatiu**: deducem daca spatiul este in sirul a sau b si refacem indicii corespunzator (daca spatiul provine din sirul a atunci decrementam indicele j, iar daca spatiul provine din sirul b decrementam indicele i
- **Nepotrivire**: aliniem caracterele diferite de pe pozitiile i din a si j din b; decrementam indicii i si j

2.3 Campuri conditionate aleatoare

2.3.1 Generalitati

2.3.2 Aplicate pe problema reconstructiei cuvintelor

Capitolul 3

Agregarea rezultatelor folosind distanța rank

Am văzut cum putem obține producții de cuvinte combinând câte o singură limbă romanică cu limba latină. Pentru a îmbunătății rezultatele vrem să folosim informația din mai multe limbi romanice moderne. Astfel, fiecare clasificator întoarce o listă ordonată de cuvinte latinești, pe prima poziție aflându-se etimonul latinesc cu cea mai mare probabilitate. Prin agregarea acestora cu o anumită metrică vom obține o lista sortata cu mai probabile cuvinte latinești. Metrica folosita este distanta rank [4] întrucat s-au obtinut rezultate bune in alte probleme de natura lingvistica precum determinarea similitudinii silabice a limbilor romanice [1], [5].

In primul rand vom defini ce inseamna o lista ordonata de elemente. In al doilea rand, vom explica distanta rank intre doua clasamente si intre un clasament si o multime de clasamente. Apoi vom prezenta o metoda de aflare a tuturor agregarilor unei multimi de mai multe clasamente folosind distanta rank. In final, vom prelucra multimea de agregari bazat pe un sistem de vot pentru a determina **o singura lista ordonata** de posibile etimoane latinesti.

3.1 Clasamente și distanța rank

Un *clasament* este o listă ordonată de obiecte după un anumit criteriu, pe prima poziție aflându-se cel cu cea mai mare importanță. În unele situații se pune problema găsirii unui clasament cât mai apropiat de o mulțime de mai multe clasamente. Pentru a rezolva această problemă trebuie sa definim mai întâi ce înseamnă distanța dintre două clasemente sau dintre un singur clasament și o mulțime de clasamente.

Există mai multe metrici folosite cu succes în diverse aplicații: distanța *Kedall tau*, *Spearman footrule*, *Levenshtein*, dar noi vom folosi distanța *rank* introdusa in articolul [4]. In intregul capitol vom folosi următoarele notatii:

- $U = \{1, 2, ..., n\}$ o mulțime finită de obiecte numită univers
- $\tau = (x_1 > x_2 > ... > x_d)$ un clasament peste universul U
- $>$ o relație de ordine strictă reprezentând criteriul de ordonare
- $\tau(x)$ = poziția elementului $x \in U$ în clasamentul τ dacă $x \in \tau$, numerotând pozițiile de la 1 începând cu cel mai important obiect din clasament

Dacă un clasament conține toate elementele din univers, atunci el se va numi *clasament total*. Asemănător, dacă conține doar o submulțime de obiecte din univers, atunci îl vom numi *clasament parțial*.

Notăm ordinea elementului x în τ astfel:

$$ord(\tau, x) = \begin{cases} |n + 1 - \tau(x)| & , x \in \tau \\ 0 & , x \in U \setminus \tau \end{cases} \tag{3.1}$$

Definiție 3.1 *Fie τ si σ două clasamente parțiale peste același univers U. Atunci distanța rank va fi*

$$\Delta(\tau, \sigma) = \sum_{x \in \tau \cup \sigma} |ord(\tau, x) - ord(\sigma, x)| \tag{3.2}$$

Se observă faptul că, în calculul distanței rank, se ia în considerare ordinea definită mai sus și nu poziția. În primul rând, cum primele poziții sunt cele mai importante, distanța dintre două clasamente trebuie sa fie cu atât mai mare cu cât diferă mai mult începutul lor.[12] În al doilea rând, definiția funcției *ord* pune accentul pe lungimea clasamentelor întrucăt putem presupune că un clasament mai lung a fost obținut în urma comparării mai multor obiecte din univers. Deci ordinea elementelor este mai solidă. Spre exemplu, dacă două clasamente de lungimi diferite au același element pe prima poziție, există totuși o diferență a ordinii obiectului în cele două liste, diferență ce contribuie la calculul distanței rank totale.[6]

3.1.1 Agregări cu distanța rank

O *agregare de clasamente* reprezintă un singur clasament σ astfel încât o anumită metrică de la acesta la mulțimea de liste de agregat T este minimă. Raportandu-ne la distanța rank avem[4]:

Definiție 3.2 *Fie un set de clasamente $T = \{\tau_1, \tau_2, ..., \tau_m\}$ dintr-un univers U și $\sigma = (\sigma_1 > \sigma_2 > ... > \sigma_k)$ un clasament astfel încât $\sigma_i \in U, \forall 1 \leqslant i \leqslant k$. Definim distanța rank de la σ la T astfel:*

$$\Delta(\sigma, T) = \sum_{i=1}^{m} \Delta(\sigma, \tau_i) \tag{3.3}$$

Definiție 3.3 *Se numește multime de agregari de lungime k a mulțimii T folosind distanța rank, setul* $A(T,k) = \{\sigma = (\sigma_1 > \sigma_2 > ... > \sigma_k) | \sigma_i \in U, \forall 1 \leqslant i \leqslant k$, *si* $\Delta(\sigma, T)$ *este minim posibila* $\}$

Problemă 3.4 *Fie U un set de obiecte si* $T = \{\tau_1, \tau_2, ..., \tau_m\}$ *o multime de clasamente peste universul U. Vrem sa determinam multimea de agregari* $A(T,k)$ *pentru un k fixat.*

Contruim următoarele matrici bidimensionale $W^k(i,j)$ cu n linii și n coloane. Fiecare celulă din fiecare matrice reprezintă costul total din distanța rank de la un clasament σ, de lungime l, către o mulțime $T = \{\tau_1, \tau_2, ..., \tau_m\}$ fixată indus de plasarea elementului $x_i \in U$ pe poziția j în σ [6]. Se observă faptul că un clasament peste universul U definit mai sus poate avea lungimea maxim n. Rezultă că numărul de coloane al matricilor W^t este egal cu n.

$$W^k(i,j) = \begin{cases} \sum_{p=1}^{m} |ord(p,i) - k + j| & ,j \leqslant k \\ \sum_{p=1}^{m} |ord(p,i)| & ,j > k \end{cases} \tag{3.4}$$

Propoziție 3.5 *Distanța de la un clasament* $\sigma = (\sigma_1 > \sigma_2 > ...\sigma_k)$ *la mulțimea T este*

$$\Delta(\sigma, T) = \sum_{x_i \in U \cap \sigma} W^k(i, \sigma(x_i)) + \sum_{x_i \in U \setminus \sigma} W^k(i, k+1)$$

unde n reprezintă numărul de obiecte din univers, iar $k < n$.

Se observă faptul că, în cazul în care σ conține toate elementele din U, deci cazul $k = n$, formula se devine

$$\Delta(\sigma, T) = \sum_{x_i \in U \cap \sigma} W^k(i, \sigma(x_i)) \tag{3.5}$$

3.1.2 Reducerea la o problemă de cuplaj perfect de cost minim

Fiecare matrice W^l din secțiunea precedentă este calculată în mod independent de celelalte Deci putem determina doar o singură matrice pentru o anumită lungime fixată l. Astfel, problema se reduce la găsirea unui clasament σ ce minimizează formula (3.5). Formal:

Problemă 3.6 *Fiind dată o matrice pătratică W,* $W = (w_{i,j})_{1 \leqslant i,j \leqslant n}$ *vrem să determinăm următoarea mulțime:*

$$S = \{(i_1, i_2, ..., i_k) | (i_p \neq i_j, \forall p \neq j), (i_j \in U) \text{ și } \sum_{j=1}^{n} w_{i_j,j} \text{ este minim}\}$$

Problema de mai sus se aseamănă cu o problemă de cuplaj de dimensiune k de cost minim întrucăt vrem să formăm perechi între obiectele dintr-un univers și pozițiile unui clasament de tip agregare, iar fiecare combinație are un anumit cost. Practic $(i_1, i_2, ..., i_n)$ reprezintă o permutare a elementelor din U.

O solutie pentru a rezolva problema precedenta este aplicarea algoritmul Ungar prezentat de Khun [10]. Altfel, putem considera matricea W ca fiind o matrice de costuri intr-un graf bipartit G pe care aplicam un algoritm clasic de gasire a cuplajului maxim de cost minim (din care luam doar k perechi). Conform [16] aceasta problema poate fi rezolvata in timp polinomial $\mathcal{O}(n^3)$ construind o retea de flux cu capacitati convenabile si prin gasirea unor drumuri de augmentare minime, din punct de vedere al costului, folosind algoritmul lui Dijkstra[3].

Toate aceste rezolvari determina o singura agregare dar nu si pe toate, adica multimea $A(T, k)$ din definitia 3.3. In continure prezentam o metoda de determinare a tuturor agregarilor bazata pe gasirea tuturor cuplajelor *perfecte* de cost minim dintr-un graf, metoda prezentata in [11]. Algoritmul ruleaza intr-un timp polinomial. Particularizam problemele si algoritmii din articolul *A generalization of the assignment problem, and its application to the rank aggregation problem* [11] pentru Problema 3.6.

3.1.3 Calcularea tuturor agregărilor optime

Reamintim faptul ca dorim sa calculam multimea de agregari $A(T, k)$, stiind costul plasarii fiecarui element pe fiecare pozitie, memorat in matricea W^k calculata la (3.4). Reformulam problema in elemente de teoria grafurilor. Astfel, asociem Problemei 3.6 un graf $G = (V, E, c, w)$, unde V reprezintă multimea de noduri, E este multimea de muchii iar $c\colon E \to \mathbb{N}$ si $w\colon E \to \mathbb{N}$ reprezinta capacitatea unei muchii respectiv costul acesteia. Legaturile intre Problema 3.6 si graful G sunt:

- $V = \{src, dst\} \cup U \cup \{1, 2, ..., k, k+1\}$
- $E = \{(src, x_i) | x_i \in U\} \cup \{(x_i, j) | x_i \in U \text{ si } j = 1, ..., k\} \cup \{(j, dst) | j = 1, ..., k\}$
- $c(muchie) = 1, \forall muchie = (x, j) \in E, j \neq k+1$
- $c(muchie) = \infty, \forall muchie = (x, k+1) \in E$
- functia w astfel:
  - $w((src, x_i)) = 0, \forall x_i \in U$
  - $w((x_i, j)) = W^k(i, j), \forall x_i \in U, j = 1, ..., k+1$
  - $w((j, dst)) = 0, j = 1, ..., k+1$

Potrivirea unui element x cu pozitia $k+1$ va semnifica excluderea acestuia din agregare ce afecteaza distanta rank dintre un clasament ce nu contine elementul x si multimea intreaga T. Se poate calcula usor in acest graf un flux maxim de cost minim folosind algoritmi clasici[16]). Notam prin $solve(W)$ un asemenea algoritm. Fie solutia $M = \{(x,j) | x \in U \text{ si } j = 1,...,k\}$. Urmatorul pas este aflarea unei solutii M' diferite de M.

Propoziție 3.7 *Doua cuplaje M si M′ sunt diferite daca exista cel putin o pereche (x,y) care se afla in M si nu se afla in M′.*

Astfel, propunem urmatorul algoritm, adaptat din [11], prin care cautam o a doua solutie M' fixand cate o muchie candidat (x,y) prin care M' sa difere de M. Setand costul muchiei (x,y) pe o valoare infinita, avem garantia ca aceasta nu va fi luata in considerare in constructia lui M'.

Algoritm 1 anotherSolution

Input: W, M
Output: M'

$s \leftarrow \sum_{(u,v) \in M} w_{uv}$
for all $(x,y) \in M$ **do**
  $temp \leftarrow w_{xy}$
  $w_{xy} \leftarrow \infty$
  $M' \leftarrow solve(W)$
  if $M' \neq \emptyset$ si $\sum_{(u,v) \in M'} w_{uv} = s$ **then**
    return M'
  else
    $w_{xy} \leftarrow temp$
  end if
end for
return $\emptyset$

Algoritmul returneaza fie multimea vida, fie o solutie M' astfel incat exista o pereche $(x,y) \in M \setminus M'$. Se poate imparti problema initiala in doua subprobleme disjuncte P_1 si P_2:

- P_1: multimea tuturor cuplajelor ce contin muchia (x,y)
  In acest caz fortam pastrarea perechii in solutie prin setarea tuturor celorlalte valori de pe linia x, coloana y pe o valoare infinita in matricea W: $w_{iy} = w_{xj}, \forall i = 1,..,n,\ i \neq x$ si $j = 1,...,n,\ j \neq y$
- P_2: multimea tuturor cuplajelor ce **nu** contin muchia (x,y)
  In acest caz perechea (x,y) nu va mai fi niciodata aleasa intr-o solutie daca costul acesteia este infinit: $w_{xy} = \infty$

Evident, exista deja cate o solutie calculata pentru cele 2 subprobleme si anume $M \in P_1$ si $M' \in P_2$. Prim urmare, se poate aplica Algoritmul 1 in mod recursiv pentru fiecare dintre aceste subprobleme pentru a determina intreaga multime de solutii. Aceasta abordare conduce la construirea unei structuri de cautare arborescente in care radacina reprezinta problema initiala 3.6, iar fiecare nod intern constituie o impartire pe subprobleme dupa o pereche (x, y). Solutia finala se construieste traversand arborele in adancime si pastrand toate solutile parțiale calculate la fiecare pas. Nu se va genera aceeasi solutie de mai multe ori prin faptul ca problemele P_1 si P_2 sunt complet disjuncte.

Algoritm 2 dfsAgregare

Input: S, M, W

$s \leftarrow \sum_{(u,v) \in M} w_{uv}$
$M' \leftarrow another_{s}olution(W, M)$
if $M' \neq \emptyset$ **then**
  return
else
  $S \leftarrow M'$
  $(x, y) \in M \setminus M'$
  $w_{xy} \leftarrow \infty$
  $dfsAgregare(S, M', W)$
  $w_{iy} \leftarrow w_{xj}, \forall i = 1, .., n,\ i \neq x$ si $j = 1, ..., n,\ j \neq y$
  $dfsAgregare(S, M' \setminus (x, y), W)$
end if

Algoritm 3 Calculeaza toate cuplajele perfecte de cost minim

Input: W
Output: S

$S \leftarrow \emptyset$
$M \leftarrow solve(W)$
$S \leftarrow S \cup M$
$dfsAgregare(S, M, W)$
return S

Algoritmul 3 determina toate cuplajele perfecte de cost minim pornind de la o matrice de costuri W. Calculand matricea W dupa formula (3.4), atunci multimea S determinata de algoritm este chiar solutia cautata in Problema 3.6. Cateva aspecte legate de complexitatea metodei prezentate:
Fie $x = |S|$, numarul total de solutii pentru o anumita problema.

- Pentru fiecare solutie nou calculata, se construiesc doua noi alte probleme. In total se vor rezolva maxim $2 * x + 1$ subprobleme.

- O subproblema necesita gasirea unui cuplaj de cost minim ce se poate calcula intr-un timp polinomial folosind metoda Ungara[10] ori un algoritm clasic de determinare a fluxului maxim de cost minim intr-un graf bipartit[16].

Intuitiv, putem afirma ca Algoritmul 3 ruelaza intr-un timp polinomial. Complexitatea acestuia a fost demonstrata in [11] ca fiind $\mathcal{O}((2x+1)k^3 n \log(n+k))$.

Subproblemele sunt complet independente de complementarele lor. Prin urmare, rezolvarile acestora se pot rula in paralel pe mai multe fire de executie. In experimentele rulate am ales sa pornim un nou fir de executie pentru fiecare subproblema de tipul P_2 pana la un anumit nivel din arborele de cautare determinat in functie de procesorul folosit. Subproblema de tipul P_1 a ramas sa ruleze pe firul de executie principal.

3.2 Determinarea tuturor agregărilor producțiilor de cuvinte

În capitolul precedent am prezentat o metoda de a combina o limbă romanică modernă si limba latină pentru a automatiza procesul de determinare a etimonului latinesc. Metoda returna primele n cuvinte posibile ordonate de la cel cu probabilitatea cea mai mare la cel cu probabilitatea cea mai mică. Vom considera aceste liste de cuvinte ca fiind clasamente. Pentru fiecare cuvânt latinesc vom agrega clasamentele produse din fiecare limbă romanică modernă *(ro, it, fr, es, pt)*. Se observă faptul că pot exista mai multe astfel de agregări așa că ne propunem să le aflăm pe toate într-un mod eficient din punct de vedere al complexității timp. Alegem să luăm în considerare doar primele 5 cele mai bune cuvinte din fiecare set. Astfel, pentru un singur cuvânt latinesc, vom avea:

$$R = \{r_1, r_2, ..., r_k\}, \text{clasamentele produse din ro, it, fr, es, pt}$$
$$k = |R|, 1 \leqslant k \leqslant 5$$
$$U = \bigcup_{i=1}^{k} r_i \text{ universul de cuvinte}$$
$$n = |U|$$

Definim o matrice bidimensională de k linii și n coloane în care calculăm ordinea fiecărui cuvânt din universul unui cuvant în fiecare clasament dat:

$$ord[i][j] = \begin{cases} |6 - r_i(x_j)| & , x_j \in r_i \\ 0 & , x_j \in U \setminus r_i \end{cases}$$

Calculam apoi matricea de costuri W^5 conform formulei de la 3.4 pentru a afla cum este afectat costul final daca un anumit cuvant $x_i \in U$ este plasat pe pozitia j. Aceasta poziitie poate sa fie mai mare decat 5 dar conform formulei construite, nu se va face nicio distinctie intre a plasa un cuvant pe pozitia 6 spre exemplu, sau pozitia 7, considerandu-se ca respectivul element nu va apartine in agregarea finala de lungime 5.

Este limpede ca se poate aplica acum Algoritmul 3 prezentat in sectiunea precedenta pentru a afla multimea $S = \{\sigma_1, \sigma_2, ..., \sigma_t\}$ de agregari posibile. Propunem combinarea solutiilor din S printr-un sistem de vot. Fiecare cuvant va primi un scor bazat pe pozitiile pe care se afla in clasamentele din S.

$$scor\colon U \to \mathbb{N}$$
$$scor(x_i) = \sum_{j=1}^{|S|} \sigma_j(x_i), \forall x_i \in U$$

Putem in sfarsit sa construim lista finala, introdusa vag la inceputul capitolului, de cuvinte ordonate in functie de probabiliatea de a fi etimonul latinesc al unor anumite cuvinte moderne din *ro, it, fr, es, pt*. Pe prima pozitie se va afla cuvantul produs cu scorul cel mai mic, adica cel ce se afla pe primele pozitii in clasamentele S, pe a doua pozitie, cel cu urmatorul cel mai mic scor, si asa mai departe. In cazul in care exista mai multe cuvinte cu acelasi scor, le vom pastra pe toate pe aceeasi pozitie si le vom filtra manual folosind reguli lingvistice.

$$F = (x_1 > x_2 > x_3 > x_4 > x_5)$$
$$x_1 = \min(scor(x))$$
$$x_2 = \min_{x, x \neq x_1}(scor(x))$$
$$x_3 = \min_{x, x \neq x_1, x_2}(scor(x))$$
$$x_4 = \min_{x, x \neq x_1, x_2, x_3}(scor(x))$$
$$x_5 = \min_{x, x \neq x_1, x_2, x_3, x_4}(scor(x))$$
$$\{x_1, x_2, x_3, x_4, x_5\} \subseteq U$$

Capitolul 4

Rezultate

4.1 Evaluare autoamata

4.2 Evaluare manuala

Capitolul 5

Concluzii

Bibliografie

[1] A.Dinu and L.P. Dinu. On the syllabic similarities of romance languages. *LNCS, 3406*, pages 785–788, 2005.

[2] A.M. Ciobanu and L.P. Dinu. Ab Initio: Automatic Latin Proto-word Reconstruction. *COLING*, pages 1604–1614, 2018.

[3] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*, chapter 24, pages 504–507. MIT Press and McGraw-Hills, 2 edition, 2001.

[4] L.P. Dinu. On the classification and aggregation of hierarchies with different constitutive elements. *Fund. Inform. 55 (1)*, pages 39–50, 2003.

[5] L.P. Dinu. Rank Distance with Applications in Similarity of Natural Languages. *Fundamenta Informaticae, 64(1-4)*, pages 135–149, 2005.

[6] L.P. Dinu and F. Manea. An efficient approach for the rank aggregation problem. *Theoretical Computer Science vol 359(1-3)*, pages 455–461, 2006.

[7] I. Fischer. *Latina Dunăreană. Introducere în istoria limbii române*. Editura Ştiinţifică şi Enciclopedică, 1985.

[8] G. Jager and P. Sofroniev. Automatic cognate classification with a Support Vector Machine. *KONVENS*, 2016.

[9] G. Kondrak. A New Algorithm for the Alignment of Phonetic Sequences. *NAACL*, pages 288–295, 2000.

[10] H.W. Kuhn. The Hungarian method for assignment problem. *Naval Res. Logist. Quarterly 2*, pages 83–97, 1955.

[11] F. Manea and C. Ploscaru. A generalization of the assignment problem, and its application to the rank aggregation problem. *Fundamenta Informaticae 81*, pages 459–471, 2007.

[12] S. Marcus. Linguistic structures and generative devices in molecular genetics. *Cahiers Ling. Theor. Appl.*, pages 77–101, 1974.

[13] S.B. Needleman and C.D. Wunsch. A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins. *Journal of Molecular Biology*, pages 443–453, 1970.

[14] T. Rama. Siamese convolutional networks for cognate identification. *COLING*, pages 1018–1027, 2016.

[15] T. Rama, J-M. List, J.Wahle, and G. Jager. Are automatic methods for cognate detection good enough for phylogenetic reconstruction in historical linguistics? *NAACL*, pages 393–400, 2018.

[16] R.Jonker and T.Volgenant. A shortest augmenting path algorithm for dense and sparse linear assignment problems. *Computing 38*, pages 5(1): 325–340, 1987.

[17] S.R. Rîpeanu. *Lingvistica romanică: Lexic, Morfologie, Fonetică*. Editura All., 2001.

[18] M. Sala. *De la Latină la Română*. Editura Univers Enciclopedic, 1998.

[19] A. Schleicher. *Compendium der vergleichenden Grammatik der indogermanischen Sprachen*. Weimar, 1861.

[20] M. Weiss. *The Routledge Handbook of Historical Linguistics*. Routledge, 2015.