**AUTOCOMPLETARE DE TEXT DEPENDENTĂ DE ISTORICUL UTILIZATORULUI**

LUCRARE DE LICENŢĂ

|  |  |  |
| --- | --- | --- |
|  | Absolvent: | **Cătălin Gabriel STAN** |
|  |  |  |
|  | Coordonatori ştiinţifici: | **Prof. Dr. Ing. Rodica Potolea**  **Conf. Dr. Ing. Mihaela DÎNȘOREANU** |

**2015**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | |  | |
| DECAN, |  | | DIRECTOR DEPARTAMENT, | |
| **Prof. dr. ing. Liviu MICLEA** |  | **Prof. dr. ing. Rodica POTOLEA** | |

Absolvent: **Cătălin Gabriel STAN**

**AUTOTOMPLETARE DE TEXT DEPENDENTĂ DE ISTORICUL UTILIZATORULUI**

1. **Enunţul temei:** *Scopul acestui proiect este de a definii și de a crea o soluție pentru un model de date bazat pe date statistice, a apariției cuvintelor folosite de către un utilizator particular.*
2. **Conţinutul lucrării:** *Capitolul 1 – Introducere - Contextul Proiectului, Capitolul 2 – Obiectivele Proiectului, Capitolul 3 – Studiu Bibliografic, Capitolul 4 – Analiză și fundamentare teoretică, Capitolul 5 – Proiectare de detaliu și implementare, Capitolul 6 – Testare și Validare, Capitolul 7 – Concluzii, Capitolul 8 – Bibliografie.*
3. **Locul documentării**: Universitatea Tehnică din Cluj-Napoca, Departamentul Calculatoare
4. **Consultanţi**: prof. dr. ing Rodica POTOLEA

conf. dr. ing Mihaela DÎNȘOREANU

ș. l. dr. ing Camelia LEMNARU

1. **Data emiterii temei:** 18 iunie 2015
2. **Data predării:** 18 Iunie 2015

|  |  |
| --- | --- |
| Absolvent: | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |
|  |  |
| Coordonator ştiinţific: | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |

**Declaraţie pe proprie răspundere privind**

**autenticitatea lucrării de licenţă**

Subsemnatul(a)**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**, legitimat(ă) cu \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ seria \_\_\_\_\_\_\_ nr. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_   
CNP \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_, autorul lucrării \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_elaborată în vederea susţinerii examenului de finalizare a studiilor de licență la Facultatea de Automatică și Calculatoare, Specializarea \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ din cadrul Universităţii Tehnice din Cluj-Napoca, sesiunea \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ a anului universitar \_\_\_\_\_\_\_\_\_\_, declar pe proprie răspundere, că această lucrare este rezultatul propriei activităţi intelectuale, pe baza cercetărilor mele şi pe baza informaţiilor obţinute din surse care au fost citate, în textul lucrării, şi în bibliografie.

Declar, că această lucrare nu conţine porţiuni plagiate, iar sursele bibliografice au fost folosite cu respectarea legislaţiei române şi a convenţiilor internaţionale privind drepturile de autor.

Declar, de asemenea, că această lucrare nu a mai fost prezentată în faţa unei alte comisii de examen de licenţă.

In cazul constatării ulterioare a unor declaraţii false, voi suporta sancţiunile administrative, respectiv, *anularea examenului de licenţă*.

|  |  |  |
| --- | --- | --- |
| Data  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |  | Nume, Prenume  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |
|  |  |  |
|  |  | Semnătura |

Cuprins

[Capitolul 1. Introducere – Contextul proiectului 1](#_Toc422324259)

[1.1. Scurt Istoric 1](#_Toc422324260)

[1.2. Exemple de situații ce necesită autocompletare 1](#_Toc422324261)

[1.3. Descrierea autocomopletării 1](#_Toc422324262)

[1.4. Funcționalitățile unui sistem de autocompletare 2](#_Toc422324263)

[1.5. Autocompletarea în contextul web 2](#_Toc422324264)

[1.6. Tipuri de autocompletare 3](#_Toc422324265)

[1.7. Provocările construirii unui sistem de autocompletare 3](#_Toc422324266)

[Capitolul 2. Obiectivele Proiectului 5](#_Toc422324267)

[2.1. Scopul proiectului 5](#_Toc422324268)

[2.2. Cerințe funcționale 5](#_Toc422324269)

[2.3. Cerințe tehnice 6](#_Toc422324270)

[2.4. Domeniu de folosință 6](#_Toc422324271)

[2.5. Tratarea problemelor. 7](#_Toc422324272)

[2.6. Folosința completărilor 7](#_Toc422324273)

[Capitolul 3. Studiu Bibliografic 9](#_Toc422324274)

[3.1. Scurt istoric 9](#_Toc422324275)

[3.2. Analiză a autocompletării ca domeniu 9](#_Toc422324276)

[3.3. Căutarea in autocompletare 10](#_Toc422324277)

[3.4. Toleranța erorilor în autocompletare 11](#_Toc422324278)

[3.5. Autocompletare dependentă de context 12](#_Toc422324279)

[3.6. Prezicerea timpurie în autocompletările de tip WSE 14](#_Toc422324280)

[3.7. Autocompletări de fraze 15](#_Toc422324281)

[Capitolul 4. Analiză şi Fundamentare Teoretică 17](#_Toc422324282)

[4.1. Ce se întâmplă în autocompletare 17](#_Toc422324283)

[4.2. Când se folosește autocompletarea 17](#_Toc422324284)

[4.2.1. La nivel de cuvânt 17](#_Toc422324285)

[4.3. Ce date folosește autocompletarea 19](#_Toc422324286)

[4.3.1. Cuvinte în autocompletarea la nivel de cuvânt 19](#_Toc422324287)

[4.3.2. Weight-ul cuvintelor 19](#_Toc422324288)

[4.3.3. Actualizarea weight-urilor 21](#_Toc422324289)

[4.3.4. Ngrame în autocompletarea de fraze 22](#_Toc422324290)

[4.4. Structuri de date folosite pentru memorarea datelor 23](#_Toc422324291)

[4.4.1. Arbori și avantajele lor 23](#_Toc422324292)

[4.4.2. Heap-uri în memorarea dicționarului 24](#_Toc422324293)

[4.4.3. Structura de date de tip ternary search tree pentru autocompletare 25](#_Toc422324294)

[4.4.4. Grafuri Orientate 28](#_Toc422324295)

[4.4.5. Grafuri în loc de suffix trees 29](#_Toc422324296)

[4.4.6. Extragerea ngramelor din graf 29](#_Toc422324297)

[4.5. Filtrarea datelor 30](#_Toc422324298)

[4.5.1. Stop words 30](#_Toc422324299)

[4.5.2. Filtrarea ngramelor și a cuvintelor 30](#_Toc422324300)

[4.6. Arhitectura generală a mecanismului de autocompletare 31](#_Toc422324301)

[4.6.1. Input - Output 31](#_Toc422324302)

[4.6.2. Module ce țin structuri de date de tip căutare. 32](#_Toc422324303)

[4.6.3. Module ce țin structuri de date de tip indexare și stocare 34](#_Toc422324304)

[4.6.4. Module ce realizează interogările. 34](#_Toc422324305)

[4.6.5. Module ce filtrează datele de input 34](#_Toc422324306)

[4.6.6. Module ce se ocupă cu construcția datelor de căutare 35](#_Toc422324307)

[4.6.7. Module ce se ocupă cu actualizarea weight-urilor cuvintelor 35](#_Toc422324308)

[4.6.8. Module ce se ocupă cu inițializarea și distrugerea sistemului 36](#_Toc422324309)

[4.7. Design Pattern-uri folosite în arhitectură 36](#_Toc422324310)

[4.7.1. High Coesion and Low Coupling. 36](#_Toc422324311)

[4.7.2. Controller 38](#_Toc422324312)

[4.7.3. Composite 38](#_Toc422324313)

[4.7.4. Strategy 39](#_Toc422324314)

[4.7.5. Observer 39](#_Toc422324315)

[4.8. Flow-uri funcționale 39](#_Toc422324316)

[4.8.1. Autocompletarea de text la nivel de cuvânt 40](#_Toc422324317)

[4.8.2. Autocompletarea de text la nivel de frază 41](#_Toc422324318)

[4.8.3. Invățarea de nou dicționar 41](#_Toc422324319)

[4.8.4. Adăugarea de noi fișiere la un dicționar existent 44](#_Toc422324320)

[Capitolul 5. Proiectare de Detaliu si Implementare 45](#_Toc422324321)

[5.1. Tehnologii 45](#_Toc422324322)

[5.2. Configurarea sistemului 45](#_Toc422324323)

[5.3. Funcționarea algoritmilor de căutare a completărilor 47](#_Toc422324324)

[5.3.1. Completarea de cuvânt 47](#_Toc422324325)

[5.3.2. Completarea de frază. 50](#_Toc422324326)

[5.4. Implementarea modulelor sistemului 51](#_Toc422324327)

[5.4.1. Input - Output 51](#_Toc422324328)

[5.4.2. Module ce țin structuri de date de tip căutare. 52](#_Toc422324329)

[5.4.3. Module ce țin structuri de date de tip indexare și stocare 55](#_Toc422324330)

[5.4.4. Module ce realizează interogările. 56](#_Toc422324331)

[5.4.5. Module ce filtrează datele de input 58](#_Toc422324332)

[5.4.6. Module ce se ocupă cu construcția datelor de căutare 59](#_Toc422324333)

[5.4.7. Module ce se ocupă cu actualizarea weight-urilor cuvintelor 60](#_Toc422324334)

[5.4.8. Module ce se ocupă cu inițializarea și distrugerea sistemului 63](#_Toc422324335)

[5.5. Vizualizarea datelor 63](#_Toc422324336)

[Capitolul 6. Testare şi Validare 65](#_Toc422324337)

[6.1. Scopul testelor 65](#_Toc422324338)

[6.2. Unități de măsură 65](#_Toc422324339)

[6.3. Descrierea modului de testare 66](#_Toc422324340)

[6.4. Descrierea seturilor de date 67](#_Toc422324341)

[6.5. Rezultate 68](#_Toc422324342)

[6.6. Interpretarea rezultatelor 71](#_Toc422324343)

[Capitolul 7. Manual de Instalare si Utilizare 73](#_Toc422324344)

[7.1. Instalare și compilare 73](#_Toc422324345)

[7.2. Utilizare 73](#_Toc422324346)

[Capitolul 8. Concluzii 75](#_Toc422324347)

[8.1. Recapitulare a ce reprezintă autocompletarea 75](#_Toc422324348)

[8.2. Contribuții și cercetare proprie 75](#_Toc422324349)

[8.3. Testare 78](#_Toc422324350)

[8.4. Dezvoltări ulterioare 78](#_Toc422324351)

[Bibliografie 79](#_Toc422324352)

# Introducere – Contextul proiectului

## Scurt Istoric

Autocompletarea, o funcționalitate de căutare care oferă sugestii pentru termenii de căutare ce utilizatorul îi scrie într-un search box pe un web site, a devenit omniprezentă pe toate motoarele de căutare sau orice site care are date indexate.

Autocompletarea de text pentru site-urile web a debutat ca și ”Google Sugest” în anul 2004, fiind privită ca o funcționalitate nu neapărat vitală, ci mai degrabă plăcută de avut într-o aplicație.

Autocompletarea de text există în mediul offline de foarte mulți ani. Primele exemple de autocompletare sunt din linia de comandă a shell-ului de UNIX, în care, la apăsarea tastei tabulator, o listă cu toate comenzile disponibile în acea locație se afișa pe ecran, sau textul se completa dacă această listă conținea un singur element.

Inițial, autocompletarea a fost concepută ca un software ce ajută la prezicerea de cuvinte ce avea ca obiectiv ajutarea persoanelor cu dizabilități fizice să își mărească viteza de scriere la un calculator, cât și să ajute la descreșterea numărului de taste necesare să scrie un cuvânt sau o propoziție. Această funcționalitate s-a dovedit utilă nu numai pentru aceste tipuri de persoane, dar pentru orice utilizator al unui calculator.

## Exemple de situații ce necesită autocompletare

Un domeniu important în care autocompletarea joacă un rol important este programarea. Programatorii din ziua de astăzi scriu foarte puține lucruri fără ajutorul unui program software de dezvoltare. Acest program are ca principal feature asistența la scriere a programatorului, pentru a elimina cele mai frecvente cauze de erori de compilare: scrierea gresită (typos).

Alți utilizatori mai importanți ca programatorii sunt persoane care sunt nevoie să scrie termeni tehnici foarte complicați și/sau lungi, cu o scriere foarte grea a acestora. Un exemplu relevant este domeniul medical. Un sistem de autocompletare ajută medicii să scrie mai bine, mai corect și mai repede rețete, diagnostice, consultații, etc.

Ca un alt exemplu, Reamintim asistarea persoanelor cu dizabilități în scriere.

Aceste exemple sunt situații în care autocompletarea trebuie să scutească utilizatorul din a scrie, cât de mult posibil, pentru a face scrisul mai rapid și/sau pentru a face scrisul mai corect.

## Descrierea autocomopletării

Autocompletarea, sau completarea de cuvinte, funcționează astfel încât, atunci când scriitorul scrie prima literă sau primele litere dintr-un cuvânt, programul prezice una sau mai multe cuvinte pe care scriitorul dorește să le scrie. Dacă cuvântul pe care dorește să îl scrie se regăsește în listă, acesta poate să îl selecteze, de exemplu folosind butoanele de litere. Dacă cuvântul pe care scriitorul îl vrea nu este prezis, atunci va trebui să continue și să introducă următorul caracter din cuvânt. În acest moment, variantele de completare sunt alterate astfel încât cuvintele sugerate încep cu literele scrise până acum de scriitor. Când cuvântul pe care scriitorul îl vrea apare în sugestii, este selectat și inserat în text.

## Funcționalitățile unui sistem de autocompletare

Un sistem de autocompletare ajută la 5 funcțioanlități [[1](#War12)]:

1. Corectarea ortografiei. Funcționalitatea de autocompletare poate să sugereze utilizatorului că un anumit termen este scris greșit. Acest lucru este foarte folositor deoarece utilizatorul poate greși ușor ortografia unor cuvinte complicate, cum sunt termenele medicale, sau variabilele într-un program. De asemenea, ajută la construcția confidenței utilizatorului într-un domeniu nou în care nu cunoaște anumite denumiri sau termeni.
2. Localizarea de idei cunoscute. În cazul în care un utilizator dorește să caute informații despre un topic, dar cunoaște doar părți despre acel topic. Această funcționalitate se combină mai bine cu prezicerea de cuvinte decât cu autocompletare, dar este o parte componentă a autocompletării.
3. Îmbunătățește certitudinea. Faptul că unui utilizator îi sunt returnate anumiți termeni sau denumiri îl asigură că acei termeni există și au fost folosiți și de alte persoane. Multe din căutările unui utilizator (în ceea ce privește căutările într-un motor de căutare) sunt completate de căutările folosite de alți utilizatori.
4. Viteza de căutare. Viteza este un factor foarte important în autocompletare. Un sistem folositor de autocompletare trebuie să returneze sugestii la fiecare apăsare de tastă de către utilizator. În cazul în care utilizatorul este forțat să aștepte după algoritmul de căutare să returneze sugestii poate crea experiența de căutare să fie una neplăcută și să determine utilizatorul să nu o folsoească atât timp cât el scrie mai repede decât aplicația returnează sugestii. Viteza de căutare se poate referi și la cât de ușor este pentru un utilizator să selecteze completări corecte sugerate de aplicație.
5. Concentrarea pe idee. Autocompletarea poate ajuta utilizatorul să își completeze ideea, nu doar cuvântul pe care îl caută. Autocompletarea poate oferi căi de completare la care utilizatorul nu s-ar fi gândit de unul singur. Acest lucru poate merge în două direcții. Constrângerea domeniului de căutare a unui termen, în cazul în care utilizatoru știe ce caută, sau lărgirea orizonturilor pentru utilizatorii care sunt la inceputul unei idei și nu sunt siguri despre cum ar trebui să scrie.

## Autocompletarea în contextul web

În contextul unui WSE (Web Search Engine), autocompletarea de interogări (search query autocompletion) este procesul de a calcula în timp real și de a sugera utilizatorului cuvinte sau fraze care pot completa interogarea pe care utilizatorul a început să o scrie, bazându-se pe interogări de utilizatori înregistrate în trecut [[2](#Kas10)]. Completările sugerate sunt folositoare deoarece în mule cazuri utilizatorul nu știe ce cuvinte să folosească sau cum să descrie informația de care are nevoie. În afară de asta, acesastă funcționalitate oferă utilizatorului șansa de a afla, fără un efort suplimentar, ce este popular în căutări pe internet pornind de la interogarea scrisă de el. Se observă așadar că autocompletările de tip web sunt în mare parte de tip explorator, și mai puțin pentru a reduce numărul de taste necesare scrierii de text.

## Tipuri de autocompletare

Autocompletarea este de două tipuri: la nivel de cuvânt și la nivel de grup de cuvinte, sau altfel spus fraze.

Chiar dacă par că merg mână în mână, cele două tipuri de autocompletare diferă prin scopul lor efectiv.

Scopul autocompletării de cuvânt este de a determina sau de a prezice, în contextul dat, ce cuvânt este cel mai probabil de a fi folosit de către utilizator. Rolul principal al acestui tip de autocompletare este acela de a reduce numărul de taste apăsate de utilizator atunci când el știe ce vrea să scrie.

Scopul autocompletării de fraze este de a determina sau de a prezice, în contextul dat, ce secvență urmează textului scris până în momentul de față de către utilizator. Rolul principal al acestui tip de autocompletare este acela de a sugera direcții de continuare a scrierii pentru utilizator, direcții folosite de alți utilizatori sau grupări de cuvinte ce au loc des.

Nu este sugerată combinarea celor două tipuri de autocompletare, datorită scopului diferit pe care acestea îl tratează. Când vine vorba despre căutări pe internet, o autocompletare de fraze este autocompletarea indicată, așa cum s-a specificat în secțiunile 1.4 și 1.5. Când vine vorba despre scrierea de mesaje pe un telefon mobil, atunci când utilizatorul știe ce vrea să zică și vrea să zică acest lucru cât mai repede posibil, o autocompeltare de cuvinte este autocompletarea indicată. Este mai orientată pe nevoile utilizatorului și dă performanțe mai bune când vine vorba de cât de utilă a fost utilizatorului.

## Provocările construirii unui sistem de autocompletare

Cea mai simplă implementare a unui sistem de autocompletare este o colecție de șiruri de caractere și interogarea acestora pentru cele care se aseamănă cu prefixul scris de utilizator. O astfel de aplicație este sortită eșecului încă de la prim început. Aceste șiruri de caractere au diferite probabilități cu care pot să apară, căutarea trebuie să se facă rapid, fără să încurce utilizatorul, sistemul ar trebui să țină cont de contextul în care utilizatorul scrie anumiți termeni. După ce utilizatorul scrie ”avion”, atunci când introduce litera ”p” o sugestie bună este ”planare” nu ”președinte”. Aceste dependențe se numesc dependențe de context, și sunt o problemă pe larg dezbătută în literatură. Un alt impediment este istoricul utilizatorului. Dacă utilizatorul scria în urmă cu un an articole științifice, dar acum scrie romane de dragoste, sistemul ar trebui să fie ”sensibil” la astfel de schimbări.

# Obiectivele Proiectului

## Scopul proiectului

Scopul acestui proiect este de a definii și de a crea o soluție pentru un model de date bazat pe date statistice, a apariției cuvintelor folosite de către un utilizator particular.

Soluția este concentrată asupra a ce scrie utilizatorul și când. Mediul în care autocompletarea este necesară poate avea informații doar despre cuvintele pe care utilizatorul le folosește. Această soluție se apropie de acest tip de probleme. Se mai numește și autocompletare offline, deoarece nu este prezent un context de web ajutător pentru determinarea de autocompletări corecte.

Pentru a valida soluția, trebuie să definim ce înseamnă autocompletarea utilă pentru utilizator. Autocompletarea presupune returnarea de sugestii de cuvinte pentru text inceput de utilizator. Scopul este reducerea apăsărilor de taste, creșterea vitezei de scriere, asigurarea că textul scris este un text bun și oferirea de sugestii de completare pentru a completa ideile utilizatorului.

Sistemul trebuie să suporte atât autocompletarea la nivel de frază cât și autocompletarea la nivel de cuvânt.

## Cerințe funcționale

Un sistem de autocompletare are 5 funcționalități [[1](#War12)]:

1. Corectarea ortografiei. Acest lucru este prezentat în [[3](#Cha09)]. Concluzia lor este că suportabilitatea unei astfel de funcționalități nu se merită când este folosită împreună cu arbori de căutare.
2. Localizarea de idei cunoscute. Această funcționalitate se rezolvă cu ajutorul autocompletării de frază.
3. Îmbunătățește certitudinea. Această funcționalitate este urmărită să fie acoperită de datele componente ale unui dicționar implicit. Cuvinte nefolosite de utilizator. Dacă le vede fiind componente din dicționarul implicit, atunci este sigur că au mai fost folosite și nu greșeste cu ceva
4. Viteza de căutare. Viteza este focusul principal al acestei lucrări. Prin construcția unei structuri de date dedicate pentru căutare rapidă și doar atât se realizează acest obiectiv. Astfel, experiența de autocompletare este plăcută, chiar dacă nu este folositoare din punct de vedere a sugestiilor returnate.
5. Concentrarea pe idee. Această funcționalitate este simulată de autocompletarea la nivel de frază.

Această soluție trebuie să fie adaptabilă după utilizator deoarece trebuie să reacționeze la cerințe/ contexte specifice utilizatorului. Acest lucru face ca implementarea să fie interactivă cu utilizatorul și să aibă capacitatea de a a-și schimba comportamentul în funcție de ceea ce utilizatorul dactilografiază.

O altă caracteristică a soluției este independența limbajului. Acest lucru înseamnă că utilizatorul poate scrie cu ajutorul diacriticelor pentru limba română (diacritice sunt: ă, â, î, ț, ș) , diacriticelor pentru limba franceză (é,è), sau tot felul de alte semne din tot felul de limbi.

Creșterea a cât de repede un utilizator scrie, poate fi măsurată din numărul de apăsări de taste scutite utiliztorului. Cele două cerințe se referă de fapt la același lucru. Cu cât este mai rapid sugerat cuvântul, cu atât utilizatorul are de introdus mai puține litere și scrie mai repede. Datorită acestui fapt, în această lucrare, focusarea este asupra cât de repede este returnat un cuvânt, sau cât de ridicat este nivelul în lista de sugestii

Indiferent de cum este construit sistemul, va dura cel puțin o tastare pentru ca sugestia autocompletării să fie selectată. Dacă utilizatorul dorește să scrie “din”, utilizatorul trebuie să scrie “d” și să apese alt buton pentru a selecta “din”. Acest lucru înseamnă două butoane apăsate plus timpul în care utilizatorul caută printre sugestii cuvântul. Sistemul nu ajută utilizatorul în cazurile în care este vorba despre cuvinte foarte scurte, această funcționalitate devenind inutilizabilă. Din aceste informații, vom continua să construim soluția pentru cuvinte mai lungi de patru caractere.

În ceea ce privește scrierea rapidă, se consideră că scrierea utilizatorului este mai rapidă și din ceea ce prinvește complexitatea selectării sugestiilor. Dacă interfața este rigidă și o sugestie este greu de selectat, utilizatorul mai mult ca sigur nu o să aleagă niciodată sugestiile returnate.

## Cerințe tehnice

Din punctul de vedere tehnic, această soluție trebuie să fie cât mai eficientă și ”ușoară” pentru mașina pe care rulează și reconfigurabilă pentru a garanta un domeniu larg de aplicare.

Pentru suportabilitatea alfabetului, utilizatorul ar trebui să poată face acest lucru cu o configurație. Pentru schimbarea funcției de descreștere a weight-urilor, utilizatorul să poată folosi o configurație. Pentru încărcarea dicționarelor dorite, utilizatorul să poată folosi o configurație.

Viteza de răspuns a unei interogări trebuie să fie de maximum 100 ms. Autocompletarea este folositoare doar când apare ”instantaneu” din punct de vedere uman, și se observă că limita superioară pentru observarea acestui fenomen este de această valoare. Dacă răspunsul sistemului este mai încet decât atât, utilizatorul o să vadă un anumit ”lag” vizual între ceea ce scrie și când apar completările. O astfel de întârziere, pe lângă că este supărătoare la ochi, face și ca sistemul să nu mai fie folosit de către utilizator.

Sistemul trebuie să consume puțină memorie și să fie scalabil. Dacă sistemul nu suportă aceste capabilități, este mai mult ca posibil să nu fie ales ca soluție pentru tratarea problemei de autocompletare.

## Domeniu de folosință

Configurabilitatea unui sistem este un prim pas spre scalabilitate și integrare în alte proiecte. Având în vedere lipsa de astfel de sisteme de autocompletare care să adreseze același scop de autocompletare offline ca și proiectul acesta, sau independența lor de context, proiectul este creat pentru a putea fi importat în orice aplicație java ce dorește să aibe o funcționalitate de autocompletare dependentă de istoricul utilizatorului.

## Tratarea problemelor.

Modelul de abordare a problemei de autocompletare este folosind frecvența de apariție a unui cuvânt. Acest mod de determinare a unui cuvânt dorit în autocompletare este cea mai rudimentară metodă de autocompletare, ea fiind prima folosită în cadrul sistemelor UNIX.

Pentru ca sistemul să poată îndeplini sarcina de dependență de istoricul utilizatorului, unui cuvânt i-au mai fost adăugate două ponderi (weight-uri) special pentru a personaliza comportamentul sistemului în ceea ce privește manevrarea cuvintelor pe care utilizatorul le folosește. Aceste weight-uri depdendente de utilizator simulează comportamentul unui cuvânt pe termen scurt și pe termen lung.

Rapiditatea returnării rezultatelor denotă necesitatea folosirii unui algoritm extrem de rapid în interogări. Această căutare trebuie să fie mai rapidă ca și căutarea clasică de parcurgere a unei liste ordonate alfabeti, sortarea cuvintelor ce încep cu prefixul precizat și returnarea primelor k cuvinte. Targetul ideal pentru această interogare este de O(k), dar până acum nu s-au găsit metode bune pentru aceste căutări care să nu depășească O(m\*n) în memorie, unde m este dimensiunea medie a unui cuvânt.

De notat este faptul că viteza de răspundere a sitemului la interogare impune o restricție stridentă asupra problemei noastre. Pentru completările de cuvânt, tehnici tipice presupun crearea unui dicționar din toate cuvintele și posibil construind acest dicționar ca un trie (sau suffix tree), cu fiecare nod reprezentând un caracter și fiecare cale rădăcină frunză un cuvânt. O asemenea tehnică poate fi folosită pentru autocompletarea de cuvinte, dar nu poate fi folosită direct în cazul multi-word (fraza) deoarece nu se poate construi un dicționar finit din aceste combinații de cuvinte. Chiar dacă limităm dimensiunile în lungime a frazelor și considerăm câteva cuvinte maximum, tot avem nevoie de un ”alfabet” format din cuvinte, iar dimensiunea unui dicționar de acest fel este de câteva ordine de magnitudine față de dicționarul format dintr-un singur cuvânt.

## Folosința completărilor

Sistemul alternează cele două completări. Când utilizatorul scrie se activează autocompletarea de cuvânt, iar după ce a introdus un cuvânt și caută să își continue ideea, autocompletarea de frază intră și ajută acest lucru. Această ciclicitate în rotirea completărilor ajută un utilizator mai bine la concentrare și întelegerea contextului în care scrie. În momentul scrierii cuvântului acesta urmărește scrierea rapidă a cuvântului, deconcentrându-se de la scris, greșind caractere. Prin faptul că nu o să mai vadă autocompletări consistente o să depisteze greșeala în scriere. După scrierea unui cuvânt, câteodată utilizatorul se gândește pentru o fracțiune de secundă ce cuvinte să folosească în continuare. Uneori acest proces poate dura și mai mult dacă utilizatorul uită un anumit termen sau dorește să folosească un sinonim (pentru a nu se repeta de prea multe ori) și nu își amintește sinonimul. În această fracțiune de secundă autocompletarea de frază aruncă utilizatorului niște posibile căi de continuare a scrisului, el poate văzând cuvântul căutat.

Oamenii au o tendință să refolosească grupuri de cuvinte, sau ”fraze” pentru a exprima înțelesuri dincolo de suma simplelor părți a acestui grup.

Un lucru subînțeles este faptul că autocompletarea este folositoare doar în situațiile în care sugestiile sugerate sunt corecte. O sugestie este considerată corectă dacă este aleasă de utilizator. Oferind sugestii care nu sunt bune este mai rău decât să nu oferi nimic, deoarece creezi anxietate și incertitudine.

Problema de adresat acum este cum se poate cuantifica faptul că un utilizator va selecta sugestia sau nu. Pot fi unele tipuri de utilizatori care scriu foarte încet la calculator și pentru ei o sugestie ce apare pe ultima poziție din listă este tot utilă. Por fi utilizatori care scriu foarte repede la un calculator și pentru doar o sugestie ce apare pe prima poziție în listă este folositoare.

Pentru a măsura acest tip de comportament, s-a introdus metrica numită precision. Scopul sistemului este de a avea precision cât mai mare.

Dacă trecem cu vederea utilizatorii experimentați de calculator, care nu sunt ținta acestui sistem, putem considera că un utilizator normal poate procesa primele 3-4 poziții din listă în timp foarte scurt și util. Un motiv este că aceste sugestii sunt aproape de poziția ochilor, iar al doilea este că un utilizator nu se chinuie să citească la fiecare apăsare de tastă sugestiile. În momentul în care una îi sare în ochi, probabil o va alege. Așadar, vom calcula și numărul de sugestii corecte ca fiind numărul de sugestii aflate în primele 4 poziții din toate interogările și vom numi acest număr usefulness.

Trecând de la cuvinte la fraze, acest usefulness este condiderabil mai greu de calculat. Având primele 5 litere dintr-un cuvânt nu este greu de determinat ce cuvinte vrea utilizatorul să folosească. Dar având primele 5 cuvinte dintr-o frază nu cuantifică nimic despre ceea ce poate să urmeze.

Ca urmare, completarea de fraze se transformă din fraze întregi într-un grup de câteva cuvinte. Așadar frază, în contextul acestei lucrări, nu se referă la o construcție gramaticală ci la o secvență arbitrară de cuvinte.

# Studiu Bibliografic

## Scurt istoric

Literatura de autocompletare este relativ nouă. Prima implementare de autocompletare folosită la nivel industrial este ”Google Suggest” lansată în 2004. Majoritatea articolelor relevante și referite în acest domeniu sunt după această dată. Înainte de 2004 se vorbea mult de algoritmi de calcul și de structuri de date (în principal arborescente) de date pentru memorarea datelor și pentru căutarea lor mai rapidă.

De la explozia internetului, multe din aceste lucruri s-au schimbat. Structurile de date care țin informații despre o interogare de autocompletare s-au diversificat de la niște simple structuri de forma <string:weight> la structuri de forma <ip\_address:date:query:weight:url:date:context:res\_num>. Contextul nu mai este reprezentat doar de structurile de cuvinte scrise în text și familii lexicale. Contextul este integrat cu site-urile accesate de un utilizator, cu reclamele pe care acesta le accesează, cu istoricul căutărilor sale pe motoarele de căutare, în multe cazuri integrat chiar și cu profilul personal de date al utilizatorului respectiv. Majoritatea site-urilor de astăzi oferă posibilitatea înregistrării pe site-ul respectiv cu conturi de Google+ sau Facebook, ceea ce dă aplicației accesul la tot ceea ce înseamnă utilizatorul respectiv. Se merge din ce în ce maimult spre centralizarea datelor unei persoane și deducerea contextului acelei persoane din alte surse indirecte. Un anumit articol publicat de un jurnalist din america (nereferențiat din motive de intimitate) declară faptul că Facebook poate contura un profil fantomă a unei persoane neînregistrate pe acest site din pozele prietenilor săi și contextul acestora.

Această creare de context se axează pe contextul de pe internet. Crearea unui context local, pe un singur calculator, este un cu totul alt domeniu.

Majoritatea articolelor din literatură axate pe acest domeniu tratează problema interogărilor motoarelor de căutare.

## Analiză a autocompletării ca domeniu

Autocompletarea ca și plug-in pentru o aplicație a devenit o funcționalitate întâlnită la tot pasul. Cercetarea pe autocompletare include o varietate de termeni tehnici care se referă la sisteme ce folosesc această arhitectură. Exemplele de menționat sunt Real Time Query Expansion(RTQE), interactive query expansion, Serach-as-you-Type(SayT), query completion, type-ahead search, auto-suggest, și suggestive searching/search suggestions. Cercetarea principală în ceea ce privește autocompletarea include probleme legate atât de arhitectura de server cât și de satisfiabilitatea utilizatorului. [[1](#War12)]

Nandi și Jagadish [[3](#Nan07)] vorbesc despre o arhitectură detaliată de sistem pentru implementarea lor de autocompletare, care sublinează principalele funcționalități și probleme întâmpinate în construcția unui index asupra căruia se v-a face interogarea. Ei observă în particular că calitatea sugestiilor prezentate utilizatorului trebuie să fie mare pentru a completa distragerea utilizatorului în momentul în care sugestiile de completare apar. Rezultatul lor se corelează cu ceea ce Hanmin spune în [[4](#Jun09)] : ”precision [of suggested terms] is closely related with satisfaction” (trad. precizia [termenilor sugerați] este strâns legată cu satisfacția [utilizatorului]). În cadrul acestei lucrări, focusul algoritmului este de a aduce această satisfacție a utilizatorului la un nivel cât mai mare. Pentru a măsura acest lucru cât mai bine, rank-ul sugestiilor este o metrică corelată direct cu acest fenomen, ca atare, în cadrul testelor, această metrică v-a fi luată în considerare.

Atât Nandi și Jagadish cît și Wu specifică facptul că un timp rapid de răspuns (sau viteză de sincronizare) este o funționalitate cheie în ceea ce privește autocompletarea, Nandi și Jagadish arătând spre o viteză de maximum 100ms. [[5](#WuH10)]

Viteza este un lucru foarte important în aplicațiile mobile, ceea ce este parte din motivul pentru care Paek et al. Recomandă autocompletarea ca parte din interfețele mobile de căutare, und ereducerea numărului de taste este o funcționalitate cheie pentru utilitate. [[6](#Pae09)]

Tot în [[1](#War12)] este explicat un studiu despre cum este folosit un sistem de autocompletare de către studenți în cadrul bibliotecii universitare pentru căutare de articole științifice. Studiul tratează ce se întâmplă în momentul în care un utilizator folosește un sistem de autocompletare și ce funcționalități din cadrul unui asemenea sistem sunt folosite.

Un sistem de autocompletare ajută la 5 funcțioanlități [[1](#War12)]:

1. Corectarea ortografiei. Funcționalitatea de autocompletare poate să sugereze utilizatorului că un anumit termen este scris greșit.
2. Localizarea de idei cunoscute. În cazul în care un utilizator dorește să caute informații despre un topic, dar cunoaște doar părți despre acel topic.
3. Îmbunătățește certitudinea. Faptul că unui utilizator îi sunt returnate anumiți termeni sau denumiri îl asigură că acei termeni există și au fost folosiți și de alte persoane
4. Viteza de căutare. Viteza este un factor foarte important în autocompletare. Un sistem folositor de autocompletare trebuie să returneze sugestii la fiecare apăsare de tastă de către utilizator.
5. Concentrarea pe idee. Autocompletarea poate ajuta utilizatorul să își completeze ideea, nu doar cuvântul pe care îl caută.

## Căutarea in autocompletare

Kastrinakis și Tzitzikas ridică problema memoriei consumate de un sistem de autocompletare pentru interogări. Ei propun ca soluție partitionarea setului de date în seturi de date mai mici [[2](#Kas10)]. Această abordare este mai scalabilă și mai rapidă deoarece încărcarea unui fragment din structura de date necesită mai puțin timp.

De asemenea se propune o modalitate de calculare a rank-ului unei sugestii în funcție de popularitatea acestea și care este prefix a altor interogări populare.

Pentru căutări, se expune faptul că algoritmul lor de căutare trebuie să întoarcă un rezultat în O(n), unde n este dimensiunea unui string căutat. Pentru aceasta se folosește un trie. Abordarea din acest articol se aseamănă oarecum cu abordarea folosită în această lucrare pentru căutare, doar că arborele ales în această lucrare are mai multe avantaje decât trie-ul, pe lângă căutarea ultra rapidă de care această structură de date o posedă.

Pentru a salva spațiu, trie-ul este împărțit în părți dependente de dimensiunea datelor care or să fie memorate în acel trie. Astfel se introduce conceptul de pădure de arbori de căutare, iar această cutare fiind delegată la arborele ce menține informația căutată. Metode de împărțire a trie-urilor au mai fost încercate și în [[7](#Jac86)] and [[8](#Bab04)], dar nu sunt potrivite pentru structura necesară autocompletării.

Partiționarea aleasă este după caracterul de început al cuvântului. Așadar, un arbore poate reține toate cuvintele din intervalul [a-k] iar celălalt din intervalul [l-z]. Bineînțeles că o descompunere așa de mică nu aduce suficiente avantaje. Analiza autorilor a arătat că pentru limba engleză, caracterele s, c și p au aproximativ 25% din cuvintele limbii engleze, pe când caracterele q,z,y și x au sub 1% combinate. Prin urmare, o distribuție uniformă este necesară.

Metoda aleasă este distribuția după primele k caractere a unui cuvânt.

Rezultatele obținute de această abordare sunt promițătoare atât din punct de vedere a timpului de interogare cât și a numărului de sugestii pe care structura de date le poate reține, îmbunătățirea fiind de 2 ordine de magnitudine.

In cadrul [[9](#Mat11)], se expune unn alt arbore și o altă metodă pentru completarea de fraze. Arborele ales este acela de SegmentTree. Mai interesant de urmărit este cum algoritmul ține minte la nivelul fiecărui nod care este weight-ul maxim din subarborele acelui nod, pentru a ști la fiecare pas încotro să meargă căutarea mai departe. Metoda de parcurgere a arborelui poate fi tradusă ca fiind un BFS cu priorități. Structura de date folosită pentru a memora nodurile rămase de parcurs este un MaxHeap în loc de o Coadă, astfel încât la fiecare pas să se parcurgă nodul cu informația cea mai ”grea” (de la weight) în subarbore. Prin această parcurgere se găsesc în timp optimal primele k sugestii de completare.

Acest articol pune cel mai bine problema a ceea ce trebuie să facă o structură de date de căutare. ” Se dă un set de n șiruri de caractere, fiecare cu un weight specific. Problema găsirii setului k cu cele mai grele șiruri de caractere cu prefixul q este problema autocompletare de frază cu rank bazată pe prefix”.

Acest articol de asemenea pune problema timpului de răspuns ce o aplicație de autocompletare găzduită pe un server ar trebui să îl aibă. Spre deosebire de [[3](#Nan07)] și [[5](#WuH10)], Matani precizează că timpul de răspuns la o interogare este de maxim 350 de ms, din care 200 sunt pentru parcurs rețeaua iar 100 pentru interfața utilizator. Rămân așadar doar 50 ms pentru procesarea serverului.

Matani desființează arobrii de tin Ternary Search Tree deoarece sunt foarte dependenți de input-ul utilizatorului, iar în cazul cel mai defavorabil, arborele se transformă într-o listă înlănțuită. Această lucrare folosește acești arbori pentru beneficiile de rapiditate în căutare, chiar dacă arborele se transformă în listă. Introducând cuvintele după weight-ul lor și arborele fiind sortat lexicografic, se asigură o căutare eficientă.

Matani mai precizează și utilitatea eliminării cuvintelor de tip stop words și a caracterelor speciale (de punctuație, spații albe, etc).

## Toleranța erorilor în autocompletare

În cadrul [[10](#Cha09)] se explică problemele ce le ridică introducerea de toleranță la eroare pentru arborii de tip trie în ceea ce privește timpul de căutare

Abordarea pentru tolerarea erorilor (de scris) este folosind metrica de asemănare între două șiruri de caractere numită edit distance. [[11](#Jur09)]

Edit distance măsoară cât de diferinte sunt două șiruri de caractere. Diferențele se măsoară în câte inserări și ștergeri trebuie efectuate pentru ca cele două șiruri să devină egale.

Rezultatele pentru incorporarea toleranței la erori demonstrează că pentru această abordare o astfel de funcționalitate nu merită să fie suportată.

* Pentru un edit distance de 0 (șiruri identice), o interogare durează 1ms.
* Pentru un edit distance de 1, o interogare durează până la 5-10ms.
* Pentru un edit distance de 3 (unde greșelile de scris se simt) timpul de căutare este în jurul a 100ms, adică depășește pragul setate de 50ms pentru partea de procesare pe server.

O altă problemă ridicată de acest articol este aceea a dimensiunii datelor de interogare. Pentru un edit distance setat la 1, se obțin următoarele rezultate

* Pentru dimensiunea de 100.000 de cuvinte în arbore, timpul de căutare este de 100 ms
* Pentru dimensiunea de 500.000 de cuvinte în arbore, timpul de căutare este de 300 ms
* Pentru dimensiunea de 1.000.000 de cuvinte în arbore, timpul de căutare este de 600 ms

Aceste valori de căutare nu sunt acceptabile. Împărțirea în subarbori (cum a făcut [[2](#Kas10)]) nu se poate aplica pe acest tip de date, deoarece căutările trebuie să străbată și arborii, iar această computație ar fi foarte costisitoare.

În urma parcurgerii acestui articol s-a decis elidarea funcționalității de corectare a ortografiei din această lucrare.

## Autocompletare dependentă de context

În [[12](#Bas06)] se vorbește despre autocompletarea dependentă de context. Bast și Weber au dezvoltat o nouă strutură de date de indexare, denumită HYB, care nu folosește mai mult spațiu decât un inderted index de ultimă generație, compresat. Această structură de date poate să răspundă autocopletărilor în câteva fracțiuni de secundă, chiar și pentru date de dimenisiuni de Terabytes.

În ceea ce privește spațiul necesar pentru HYB, s-a definit o noțiune numită empirical entropy, ce capturează complexitatea spațiului unui index independent de o schemă de compresie particulară. Se demonstrează faptul că entropia empirică a HYB-ului este în esență egală cu cea a INV (inverted index), acelasă rezultat fiind arătat și de teste.

Din punct de vedere a timpului de procesare, se oferă un număr precis de operații necesare pentru comportamentele în situațiile cele mai favorabile, defavorabile și în situația medie. Se ia în considerare până și latency-ul accesului aleator și serial a datelor.

În ceea ce primește definirea termenului de autocompletare,ei o fac diferit. O interogare de autocompltare este o pereche (D,W), unde W este un interval de cuvinte (toate posibilele completări pe care utilizatorul a început să îl scrie) iar D este un set de documente (potrivirile pentru părțile anterioare din interogare). A procesa interogarea înseamnă să se calculeze subsetul W’ al W de cuvinte ce apar în cel puțin un document din D, cât și un subset D’ al D de documente care conțin cel puțin unul din aceste cuvinte.

Inverted Indexu-ul este structura de date aleasă pentru majoritatea aplicațiilor de căutare [[13](#Wit99)]. Este relativ ușor de implementat și de extins cu alte funcționalități și are o localizare de acces excelentă.

În această lucrare, un inverted index se referă la următoarea structură de date: pentru fiecare cuvânt se memorează lista id-urilor tuturor documentelor ce conțin acel cuvânt, sortate în ordine crescătoare.

Noua structură de date, HYB are cla bază următoarea idee: precalculează listele inversate pentru uniuni de cuvinte. Considerăm o interogare de autocompletare (D,W), unde uniunea tuturor listelor pentru intervalul de cuvinte W au fost precalculate. Mulțimea D’ se obține dintr-o singură intersecție (a D cu lista precalculată). Totodată, din doar din această listă precalculatănu se poate deduce setul W’ de completări ce duc la o potrivire. Deoarece W poate fi un interval arbitrar de cuvinte, nu este clar ce uniuni de cuvinte ar trebui precalculate, în special când spațiul ce se dorește a fi folost nu trebuie să depășească spațiul folost de un inverted index.

Tot de la aceeași autori vine articolul [[13](#Bas)], care tratează aceeași problemă, dar cu o altă abordare, și anume prin construcția de arbori.

Definiția dată în acest articol se aseamănă mult cu cea anterioară, dar nu sunt amintite decât componentele D și W. O interogare de autocompletare este o pereche (D,W) unde W este un interval de cuvinte (toate autocompletările posibile ale cuvântului pe care utilizatorul îl scrie în momentul de față) și D este un set de documente (potrivirile de completări pentru părțile anterioare de interogări). Pentru a procesa interogarea înseamă să se calculeze toate seturile de perechi de cuvinte-în-documente (w,d) cu w din W și d din D.

Se definește o structură de tip AutoTree pentru aceste interogări.

Și acest articol, după cum reiese din definiția lui, se bazează pe algoritmul inverted index.

Ideea din spatele articolului este să se mărească cantitatea de prerpocesare calculând listele inversate nu doar pentur cuvinte, dar și pentru prefixele lor. Mai exact, se construiește un arbore binar de căutare cu cuvintele distincte drept frunze. Fiecare nod v-a conține ca listă de documente intersecția listelor din subarbore. Acest arbore seamănă mai ult cu un segment tree folosit de Matani în [[9](#Mat11)] decât un arbore binar de căutare, deoarece la fiecare not se memorează intervalul de informații pe care acel subarbore îl găzduiește.

Procesarea unei interogări se face în felul următor (se dau seturile W și D):

1. Calculează secvența minimală unică v1,...vl de noduri cu proprietatea că subarborii lor acoperă exact intervalul de cuvinte W. Proesează aceste l noduri de la stânga la dreapta, și pentru fiecare nod v apelează procedura de la punctul 2.
2. Preia lista Dv a v și calculează intersecția D ∩ Dv. Dacă intersecția este vidă, treci mai departe. Altfel, dacă v este o frunză corespunzătoare cuvântului w, raportează pentru fiecare d D ∩ Dv perechea (w,d). Dacă v nu este frunză, apelează procedura în mod recursiv pentru cei doi fii a lui v

Prin acest algoritm se pot tăia (prunning) foarte multe căi de căutare în arbore. Dacă un nod are intersecția vidă, atunci întregul subarbore nu va mai da nici un rezultat. Același principiu este aplicat și în această lucrare de licentă, principiul de tăiere de rezultate cât mai repede posibil.

## Prezicerea timpurie în autocompletările de tip WSE

O problemă interesantă atacată în [[15](#Bar11)] este aceea de a încerca să se facă prezicerea de completare după scrierea de un singur caracter de către utilizator. În mod evident, din primele caractere scrise este aproape imposibil să ghicești ce dorește să scrie utilizatorul.

Autorii spun un lucru interesant. Principiul de bază ce susține cele mai multe sisteme de completare este ”wisdom of the crowds”. Sugestiile sugerate de motoarele de căutare sugerează utilizatorului completări populare printre utilizatori în trecut.

În cazul în care acest lucru se poate îmbunătății (completarea de la primul caracter), experiența de autocompletare ar fi una mult mai bună, ciclul de scriere interogare -> rezultat -> rafinarea interogării s-ar scurta foarte mult. Acest lucru are ca implicare directă și salvarea resurselor computaționale.

Contextul necesar pentru a putea realiza acest lucru se compune din foarte multe lucruri. Interogări recente, pagini web vizitate recent, tweet-uri recente ale utilizatorului. Sunt doar câteva din activitățiile ce pot da un context mai clar al utilizatorului.

Conturarea contextului înseamnă pentru utilizator mai puține cuvinte, legate de domeniu, pe care el le folosește. Această idee a fost integrată în această lucrare de diplomă. Din cauză că determinarea unui context este așa complexă, simularea unui context a fost făcută prin posibilitatea schimbării dicționarului de uvinte în care caută sistemul.

După cum acest articol se axează cel mai mult pe istoricul interogărilor utilizatorului, așa și această lucrare abordează problema completării din istoricul utilizatorului.

Se introduce importanța corelației între succesiunile de interogări. La fel, în această lucrare se tratează problema integrări autocompletării de cuvinte cu autocompletarea de fraze, pentru a crea pentru cuvinte o succesiune de cuvinte ce se cunosc că apar unul după altul.

Această abordare nu este fără defecte, așa cum au observat și autorii, 89% din interogări fiind perechi noi.

Abordarea de create de clustere pare că se îndreaptă în direcția bună, dar 37% din perechile de interogări nu au mai apărut în lug-uri înainte.

Abordarea aleasă de autori este diferită. Algoritmul lor se bazează pe următoarea presupunere: când contextul este relevant utilizatorului pentr interogarea dorită, interogarea dorită este probabil să fie similară cu interogările de context. Similaritatea poate să fie sintactică sau doar semantică. După estimările lor, 56% din interogări sunt ne-sintactice.

Metodele de determinare a similitudinii unor interogări nu satisfac nevoile autorilor pentru autocompletarea căutată de ei. Exemple de moduri de similitudine sunt: reformulări, co-clicks, user search behavioral models, interogări cu rezultate similare, algoritmi de recomandări de interogări.

Noua metodă de determinare a similitudinii extinde algoritmi de recomandări de interogări. Se expandează fiecare interogare într-o reprezentare mai bogată ca și un vector de feature-uri de înaltă dimensiune li se măroară cosine similarity între reprezentările expandate.

Expandarea se face prin aplicarea iterativă a algoritmului de recomandări de interogări peste interogarea făcută, apoi peste rezultatele primei aplicări, apoi peste rezultatele celei de a doua aplicări, etc.

Se prezintă 2 algoritmi în această lucrare. NearestCompletion, care sugerează utilizatorului completări ale prefixului introdus care sunt cele mai similare cu cele mai recente interogări. Acest algoritm este folositor doar atunci când contextul este relevant pentru interogarea ce vrea să o folosească utilizatorul, rezultatele fiind cu 48% mai mare decât algoritmul standard MostPopularCompletion. S-a mai propus și algoritmul HybridCompletion, care este o combinație convexă între NearestCompletion și MostPopularCompletion. Acest algoritm este cel puțin la fel de bun ca NearestCompletion când contextul este relevant și aproape la fel de bun ca si MostPopularCompletion când contextul este irelevant.

## Autocompletări de fraze

În cadrul autocompletării de fraze [[16](#Nan071)] marile probleme întâmpinate sunt 2: numărul de fraze este combinatoric mai mare decât numărul de cuvinte, și o frază, spre deosebire de un cuvânt, nu are o frontieră bine determinată.

În domeniul de predicție de fraze s-a făcut un progres considerabil pentru limbaj natural, dar aparent foarte puțină atenție este îndreptată spre precizia acestui tip de completare, și mai degrabă focusul se duce spre structurile de date ce memorează aceste fraze. Cele mai multe modele și sisteme sunt construite pentru folosință de tip offline, și nu sunt construite pentru folosință de tip real–time, la viteza de scriere a utilizatorului uman – câteva interogări pe secundă. De asemenea, modelele de date implementate de cei mai mulți sunt ignorante în ceea ce privește probleme cum ar fi memoria de sistem.

Autorii introduc o structură de date numită FuzzyTree pentru a adresa prima problemă și conceptul de frază semnificativă pentru a doua problemă.

La nivel fizic, se propune construcția unei structuri de tip suffix tree bazată pe cuvinte numită FuzzyTree, unde fiecare nod din arbore corespunde unei instanțe de cuvânt din corp, iar căile de tip rădăcină – frunze prezic fraze. Această idee de cale în arbore și memorare a structurilor de cuvinte în acest fel se folosește și în această lucrare, cu singura diferență că se foosește un singur graf global, în loc de o pădure de arbori pentru fiecare cuvânt.

Interogările sunt executate în momentul primirii unei fraze prefix de la utilizator. Suffix-tree-ul este apoi traversat pentru a obține rezultatele completărilor. De asemenea s-a încercat miniminzarea dimenisiunilor arborilor și durata de construcție a acestora.

Din acest studiu bibliografi putem deduce că domeniul de autocompletare este încă în mare dezvoltare. Accentul în literatură se pune pe sisteme ce se folosesc în cadrul site-urilor sau pentru motoarele de căutare. Un sistem orientat pe ajutarea utilizatorului să scrie documente este sau foarte greu de găsit sau trecut foarte ușor cu privirea.

# Analiză şi Fundamentare Teoretică

În cadrul acestui capitol se vor prezenta fundamentele teoretice pe care se bazează această soluție, arhitectura soluției propuse pentru autocompletare, design pattern-urile folosite și flow-urile funcționale ale soluției.

Primele secțiuni, de la 4.1 la 4.5 se vorbește despre fundamentele teoretice. Sunt expuse soluțiile la problemele ridicate, modul de abordare a acestora, structurile de date prin funționarea lor, avantajele ce le aduc și argumentările folosirii acestora.

Secțiunile 4.6 și 4.7 descriu arhitectura sistemului ce implementează această soluție propusă pentru autocompletare. Sistemul este descompus în module, fiecare modul fiind explicat și fiind argumentată folosirea acestora. Se expun funcționalitățile, implementăriile fiind descrise în capitolul 5.

Secțiunea 4.8 descrie flow-urile funcționale și cum lucrează împreună modulele pentru realizarea funcționalităților soluției.

## Ce se întâmplă în autocompletare

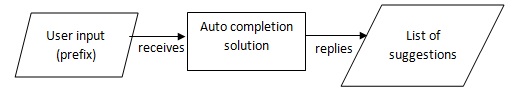


Figura 4.1. Autocompletarea ca și black box

După cum se arată în figura 3.1, o funcție de autocompletare primește date de intrare de la utilizator (prefixul cuvântului) și returnează sugestii de autocompletare. Aceste sugestii sunt cuvinte care încep cu prefixul dat de către utilizator și reprezintă cu cea mai mare probabilitate cuvintele pe care utilizatorul dorește să le scrie în contextual dat. Aceste sugestii sunt ordonate într-o listă, după probabilitățile de apariție ale fiecărui cuvânt.

## Când se folosește autocompletarea

### La nivel de cuvânt

Viteza de scriere a utilizatorului este dată de câte taste apasă acesta în timpul scrierii. După cum s-a descris în secțiunea 2, măsurăm performanțele sistemului după câte taste scutim utilizatorului în scriere.

Cu cât un cuvânt este sugerat mai rapid în autocompletare, utilizatorul scrie mai puține litere. Datorită acestui fapt, în această lucrare focusul este pe cât de repede este returnat un cuvânt (cât de ridicat este nivelul în care acesta se găsește în lista de sugestii)

Folosirea practică a unui sistem de autocompletare este când utilizatorul vrea să introducă cuvinte lungi sau cuvinte folosite foarte des. Exemple de aceste cuvinte sunt: ”pe”, ”la”, ”care”, ”exemplu”, ”pantofi”. Se observă că acele cuvinte cu frecvență mare sunt prepoziții sau cuvinte de legătură. Aceste cuvinte sunt denuminte stop words în literatură [[11](#Jur09)].De asemenea, aceste cuvinte au dimensiune mai mică decât 4 caractere.

Indiferent de cum este construit sistemul, utilizatorul are nevoie de apăsarea a cel puțin unei taste pentru ca sugestia autocompletării să fie selectată. Dacă utilizatorul dorește să scrie ”din”, trebuie apăsată tasta ”d” plus încă cel puțin o tastă pentru selecția cuvântului ”din” din lista de sugestii.

Acest lucru înseamnă cel puțin două butoane apăsate, față de 3, cât durează tastarea cuvântului în sine. Pe lângă această reducere de scriere de 1 caracter, un utilizator neexperimentat în scriere de text la calculator, care nu poate să scrie fără a privi tastatura, are nevoie de a privi ecranul calculatorului și a înțelege completările sugerate de sistem. Acest lucru durează mult mai mult decât apăsarea efectiv a tastei necesare pentru a termina de scris cuvântul.

Sistemul nu este de nici un folos utilizatorului dacă oferă autocompletare de cuvinte mai scurte de 4 caractere. Autocompletarea ar funcționa, dar utilizatorul nu va alege niciodată completările sugerate în acest caz. Un alt dezavantaj este că aceste cuvinte scurte cu weight foarte mare ar lua întâietate în lista de sugestii atunci când utilizatorul se află la începutul cuvântului. Dacă cuvântul căutat are frecvență mare și ar apărea de la prima tastă apăsată în topul sugetiilor, am încurca utilizatorul cu selectarea acestui cuvânt prin parcurgerea listei pentru a trece de stop words și a selecta cuvântul dorit.

Autocompletarea la nivel de frază este o operație care returnează foarte multe sugestii. Dacă soluția ar începe să returneze sugestii de frază încă din momentul în care începe să returneze sugestii de cuvânt, ar apărea 2 probleme:

* Ar fi foarte multe sugestii. Din cauza aceasta, ar fi foarte dificil să se aleagă doar primele k sugestii pentru a se afișa utilizatorului. Acesta nu ar vedea date complete, iar datele ce le-ar vedea ar putea fi de două feluri
  + Prea diferite. Utilizatorul ar trebui să își rezerve o bună bucată de timp pentru a citii toate sugestiile, căutând sugestia potrivită pentru el.
  + Prea asemănătoare. Sugestiile poate ar proveni toate din același cuvânt mai des folosit, iar utilizatorul să nu aibă nevoie de acest tip de sugestii. Dimpotrivă, așa cum se menționează în (Ward, Hahn, & Feist, DECEMBER 2012), autocompletarea “ajută la completarea gândurilor”. Așadar, diversitatea este încurajată și binevenită, dar când sugestiile sunt din același context.
* Ar fi foarte multe sugestii de procesat pentru utilizator. Având în vedere că utilizatorul se află la stadiul de autocompletare de cuvânt, ar fi prea multă informație de procesat pentru ca utilizatorul să mai bage în seamă aceste completări și funcționalitatea ar ajunge să fie nefolosită. O alternativă la această situație este concatenarea sugestiilor de frază peste cele de cuvânt, astfel încât la fiecare cuvânt ce apare să se pună continuarea sugestiei cele mai importante, astfel încât utilizatorul să vadă fraze care încep cu acele cuvinte. Această abordare poate funcționa așa cum funcționează pentru motoarele de căutare. S-a decis totuși să nu se meargă pe această soluție deoarece utilizatorul folosește mai mult autocompletarea de cuvânt, cea de frază fiind foarte imprecisă și dificil de înțeles.

Autocompletarea la nivel de frază nu este folosibilă in contextul nostru deodată cu autocompletarea la nivel de cuvânt. Prin urmare, autocompletarea la nivel de frază se va declanșa la introducerea unui cuvânt, și se va dezactiva când utilizatorul v-a începe să scrie alt cuvânt.

## Ce date folosește autocompletarea

### Cuvinte în autocompletarea la nivel de cuvânt

Unul din scopurile sistemului este de a funțional pentru orice nou utilizator încă de la prima tastă apăsată.

Pentru a satisface această funcționalitate, sistemul are un set de cuvinte predefinite pentru care interogările pot da rezultate. Orice cuvânt utilizat va fi tratat cu p prioritate mai mare decât cuvintele predefinite pentru ca sistemul să ”știe” ce scrie utilizatorul și ce vocabular folosește.

Soluția construiește un model de date din cuvinte. Un cuvânt, din perspectiva sistemului, este o asociație a unui șir de caractere și a unui weight(rom. pondere), descris și în formula (4.1). Weight-ul ne oferă proritatea cuvântului. Sugestiile sunt ordonate după prioritate, astfel probabilitatea unui cuvânt de a fi utilizat este în relație directă cu weight-ul pe care în are cuvântul.

O colecție de cuvinte în contextul nostru este numită dicționar. Cuvintele predefinite cu care sistemul pornește sunt grupate într-un dicționar implicit și cuvintele pe care utilizatorul le folosește sunt puse într-un dicționar utilizator. Ambele dicționare folosesc, ca și regulă pentru determinarea weight-ului, frecvența de apariție a cuvântului. Diferența dintre frecvențe este în următoarele: frecvența pentru dicționarul implicit vine din fișiere date de utilizator ca fiind fișiere de unde se pot antrena acestea (prin parcurgere și contorizare); frecvența pentru dicționarul utilizator vine chiar din ceea ce utilizatorul scrie. Dacă prioritățile weight-urilor sunt adecvate, rezultatul este mai mult sau mai puțin optimal din punct de vedere al utilizatorului.

Scrie despre structura de tip cuvant weight. Explica weight –ul

Degradarea weight-urilor

### Weight-ul cuvintelor

După cum s-a descris anterior, un cuvânt poate fi prezent atât în dicționarul utilizator cât și în cel predefinit. Acest lucru se întâmplă când un utilizator folosește un cuvânt ce apare în dicționarul implicit. Informațiile din aceste două dicționare nu sunt amestecate, dar sunt utilizate împreună pentru a calcula weight-ul global al cuvântului. Weight-ul global este calculat după formula (4.2)

Luând în considerare că aceste ponderi nu au aceeași prioritate pentru utilizator, o formulă mai exactă este formula (4.3).

Dar această formulă încă nu este completă.

Să luăm în considerare următorul caz:

Un cuvânt predefinit are o frecvență de apariție de 100 de ori (weight implicit). Un utilizator așteaptă ca un cuvânt folosit de el să devină accesibil ca primă sugestie relativ repede, din primele folosințe a lui. Pentru ca acest lucru să se întâmple, raportul utilizator (definit în formula 4.4) trebuie să fie foarte mare, de ordinul zecilor.

O astfel de discrepanță între prioritatea cuvântului folosit de utilizator și prioritatea cuvântului din dicționarul implicit nu este practică. Cuvântul în dicționarul implicit ar trebui să aibă frecvența de utilizare a acestui cuvânt în limba respectivă. Ignorarea completă a unui astfel de comportament, practic a weight-ului implicit, nu este de dorit.

#### Introducerea Actualității unui cuvânt

Soluția introduce componenta de actualitate. Această componentă reflectă utilizarea pe termen scurt a unui cuvânt, astfel încât utilizatorul poate vedea un cuvânt recent utilizat fiind returnat în topul sugestiilor foarte repede. În cazul în care cuvântul nu este folosit pentru o perioadă de timp / nu este ales dintr-un set de interogări, acesta ar trebui să scadă treptat în prioritate.

Formula pentru weight-ul global se transformă astfel în (4.5)

Atualitatea este o proprieteate dependentă de utilizator. Aceasta este memorată în dicționarul utilizatorului. Actualiatea este incrementată bazându-se pe o funcție exponențială, până când ajunge la weight-ul implicit al cuvântului cu cel mai mare weight implicit, pentru ca acest cuvânt să fie returnat în primele sugestii în momentul unei interogări. Actualitatea scade în mod similar pentru a simula lipsa utilizării într-o perioadă recentă. Astfel, ar trebui să dispară din lista de sugestii pentru interogări generale (prefixul are dimensiune de 1-2 caractere).

Un exemplu de comportament a acestei componente este descris în figura 4.2

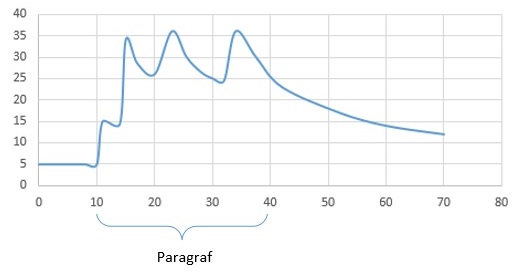


Figura 4.2 comportamentul componentei de actualitate

Figura 4.2 reprezintă un timeline a cum scrie utilizatorul. Pe axa Ox sunt weight-urile cuvintelor, iar pe axa Oy sunt reprezentate numărul de interogări la sistemul de autocompletare.

Utilizatorul are de scris un paragraf care folosește un cuvânt x de mai multe ori, iar utilizatorul este interesat să folosească acest cuvânt x doar în acest paragraf. Paragraful începe la interogarea cu numărul 10 și se termină la interogarea cu numărul 40. Înainte de începerea paragrafului, weight-ul cuvântului x este foarte mic, acesta nefiind returnat în lista de sugestii. La prima scriere a cuvântului x, sistemul vede acest lucru și crește actualitatea cuvântului x. La fiecare din următoarele interogări, actualitatea crește, dar nu depășește weight-ul maxim din structura de date. După fiecare interogare, sistemul începe să descrească actualitatea, pentru a simula încetarea folosirii de către utilizator a cuvântului. După terminarea paragrafului, cuvântul x nu mai este folosit, iar acesta o să descrească în timp la weight-ul pe care îl avea înainte de scrierea acestui paragraf.

### Actualizarea weight-urilor

Frecvența utilizatorului și actualitatea cuvântului sunt degradate în timp. Un cuvânt poate fi folosit întrun paragraf al documentului, dar după acesata utilizatorul nu mai are nevoie de el. Soluția ar trebui să respecte acest mod de comportament astfel încât un cuvânt să dipară din lista de sugestii. Acest comportament se corelează cu actualitatea utilizatorului.

Acțiunea de decrementare este numită degradare, iar regula a cât de repede această degradare trebuie să decurgă este dată de formula (4.5)

Alpha spune cât de repede funcția descrește (figura 4.3). T este variabila de timp. Sistemul nu are noțiunea de timp, doar de interogări. Așadar t este numărul de interogăr primite de soluție. Q este numit normalizator.

Graficul acestei funcții este ilustrat în figura 4.3. Alpha folost pentru acest grafic este 0.4, q este 1. Se observă că pentru ca funcția să coboare la ½ din valoarea la momentul 0, t trebuie să fie egal cu 2\*q

Parametrul q ajută la reglarea funcției pentru a putea fi ușor înțeleasă de oameni. Lucrând cu valori de 0.01 ale alpha nu este un task ușor. Spunând că după 30 de interogări valoarea funcției scade la jumătate este mai inteligibil și mai ușor de lucrat.

Valorile alese pentru q pentru implementarea noastră sunt: 100 pentru frecvența de utilizator și 15 pentru actualitate.

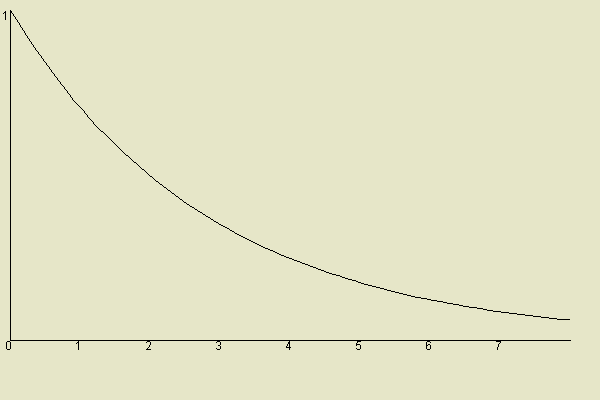


Figura 4.3. Decay function with alpha = 0.4 and q=1. On axis OY is f(t,q) and on axis OX is t/q

### Ngrame în autocompletarea de fraze

Autocompletarea la nivel de frază folosește doar strucura de <cuvânt>. Weight-ul este încă folosit de autocompletarea la nivel de frază, dar în alt mod. Vezi secțiunea 4.4.4 pentru detalii.

Cuvintele sunt grupate în ngrame. Un ngram reprezintă o “secvență continuă de n entități dintr-o secvență dată de text” (Jurafsky and Martin 2009). Un ngram de dimensiune n mai este referit ca “n-gram” (“patru-gram”, “cinci-gram”, etc).

Weight-ul se repartizează nu pe cuvânt, ci pe conexiunea între cuvinte.

**(Def)** Conexiunea între cuvintele x, y este o functie f(x,y) ce reprezintă de câte ori cuvântul y apare după cuvântul x in ngramele memorate in structura de date. (figura 4.4)

Impărțirea pe date implicite și dependente de utilizator nu se face în cadrul acestei autocompletări, deoarece structurile de cuvinte nu se schimbă într-o limbă. Utilizatorul nu inventează noi moduri de a scrie în limba respectivă, eventual folosește noi structuri neînvățate până acum de structura de date.

Un Weight-ul este asociat cu fiecare ngram. Acest weight este necesar pentru ordonarea ngramelor atunci când se v-a face returnarea sugestiilor pentru autocompletarea la nivel de frază.

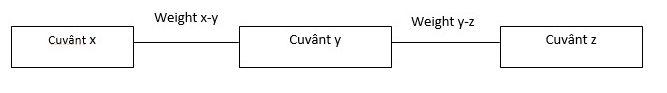


Figura 4.4. Structura unui ngram

## Structuri de date folosite pentru memorarea datelor

### Arbori și avantajele lor

Măsura de sugestii rapide și corecte necesită crearea unei structuri de căutare foarte rapidă. Căutarea într-o listă sortată alfabetic se poate face în timp logaritmic, folosind căutare binară, dar după ce se găsesc cuvintele cu prefixul dorit, trebuie efectuată ordonarea acestora după weigth. În cazul în care prefixul nu este mai lung de o literă sau două, pot fi foarte multe cuvinte de sortat. În această metodă căutarea este lentă.

O modalitate mai rapidă de căutare este de a avea cuvintele sortate după weight-uri și apoi sortate în ordine alfabetică. Se sortează cuvintele după weight și se încarcă într-un arbore. Acest arbore trebuie să respecte proprietățile arborelui de căutare binar (Binary Search Tree, i. e. BST). Un BST sortează nodurile alfabetic, deci la fel ar trebui să facă și arborele nostru de căutare.

Acum soluția are o structură de date care are acces rapid atât la ordinea alfabetică a cuvintelor cât și la ordinea weight-urilor cuvintelor. În comparație cu listele ordonate alfabetic, sortarea la fiecare căutare este eliminată.

De exemplu, dacă datele de intrare din dicționar sunt structurate astfel: ”barbie=2”, ”barber=1”, ”beach=1”, ”beer=10”, ”bollywood=1” și ”breath=9” structura va arăta ca în figura 4.5.

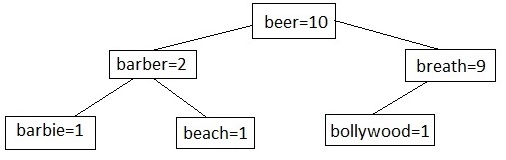


Figure 4.5. structura de arbore binar de căutare

După cum se vede în figura 4.5, căutând cuvinte care încep cu ”be” forțăm algoritmul de a merge pe toate drumurile până la frunzele arborelui în scopul de a returna atât ”beer” cât și ”beach”, deoarece cuvintele nu sunt ordoate complet alfabetic. Scopul este de a returna aceste cuvinte dintr-o zonă de căutare mai mică. Acest lucru se poate face cu un arbore care împarte cuvintele în caractere și potrivește secvențele comune. Cuvintele sunt o serie de noduri dintr-un drum. Această idee este implementată de Ternary Search Trees și arată ca în figura 4.6

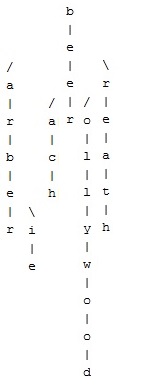


Figure 4.6 Ternary Search Tree model

Avantajele acestei reprezentări este acela că este lightning fast pentru k sugestii de cuvinte pentru un prefix dat.

Pentru a putea beneficia pe deplin de ambele structuri, modelul de date v-a fi creat din 2 părți. Partea de indexare a datelor, care va avea unicul scop de a menține dicționarul și de a actualiza ponderile cuvintelor, și partea de căutare a datelor, care va fi o structură de căutare rapidă precum propunerea de Ternary Search Trees.

### Heap-uri în memorarea dicționarului

Având în vedere descompunerea detelor în câte un arbore pentru fiecare literă din alfabet (modelul de căutare), se creează o discrepanță între datele memorate în datele de căutare și datele memorate în dicționar.

Această discrepanță are ca efect următoarele:

* Actualizarea unui cuvânt durează mai mult. Dicționarul are toate cuvintele la un loc, deci căutarea și actualizarea unui cuvânt sunt operații mai costisitoare
* Discrepanța între datele de căutare și datele din dicționar face reconstrucția datelor de căutare mai înceată. Un singur flux de date poate trece prin dicționar pentru interogări. Dacă modelul ar fi împărțit, această operație ar putea fi paralelizată foarte bine.
* Parcurgerea inutilă a cuvintelor care nu sunt target-ul unei căutări.

Datorită acestor dezavantaje, dicționarul v-a fi împărțit și el la fel ca și modelul de căutare, în grupuri de cuvinte ce încep cu aceeași literă.

Pentru memorarea dicționarului s-a ales o structură de tip heap ce memorează cuvintele sortate după weight. Această decizie este argumentată în secțiunea următoare.

#### Actualizarea dicționarului

In momentul actualizării modelului, sistemul suprascrie weight-ul cuvântului folosit în structura de căutare, dar păstrează structura la fel. Arborele nu știe cum să refacă structura în momentul decrementării weight-urilor într-un mod simplu, care nu necesită computație multă. Datele de indexare pot fi modificate cu ușurință cu orice fel de weight. Când se actualizează un weight, heap-ul care ține datele poate să mute cuvântul în sus sau în jos pe nivelurile sale. Această mutare a cuvintelor nu se reflectă și în datele de căutare, iar când se efectuează o căutare, algoritmul trebuie să parcurgă arborele mai adânc decât ar fi nevoie. Reconstrucția modelului de căutare este necesară în acest caz, pentru a menține parametrii de viteză ai căutării în valori bune.

Luând în cosiderare faptul că de fiecare dată când utilizatorul scrie un cuvânt, weight-ul cuvântului este actualizat, și fiecare interogare decrementează weight-urilor celorlalte cuvinte, structura de căutare pierde repede imaginea structurii de indexare.

Reconstrucția nu se face după fiecare interogare sau modificare de weight. Este nevoie de o funcție de calcul a acestei necesități de reconstrucție. Această funcție este numită funcția de cost și este dată de formula (4.6).

Din cauza necesității unei astfel de funcții, și imposibilitatea calculării ei cu orice altă formulă, dicționarul memorează cuvintele in heap-uri.

### Structura de date de tip ternary search tree pentru autocompletare

Sistemul folosește o structură de date de arbore care meorează cuvinte pentru a răspunde la interogări pe model. Ternary search trees combină beneficiile unei memorii efiiciente a Binary Search Trees și rapiditatea mare de căutare a arborilor digitali (Trie) de căutare.

Pentru a exemplifica aceste avanaje și dezavantaje, se vor lua ca exemplu următoarele cuvinte: in, be, as, at, by, he, of, is, it, or, on, to.

#### Bynary search tree

Figura 4.7 arată cum un arbore binar de căutare (BST) memorează aceste cuvinte. Rădăcina conține un cuvânt, și pentru fiecare nod , fiul din stânga conține un cuvânt care este mai mic decât nodul curent și fiul drept este mai mare. Căutarea prin BST este de la rădăcină în jos până când se găsește un nod cu cuvântul căutat. Pentru a găsi cuvântul, prima verificare va fi în pe nodul respectiv. Dacă este mai mare mergem în dreapta. Dacă este mai mic mergem în stânga. Dacă este găsit ne putem opri.

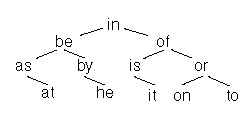


Figura 4.7 Arbore binar de căutare

#### Digital trie

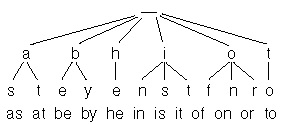


Figura 4.8 Digita Trie

Figura 4.8 arată cum un arbore digital memoriază cuvintele. Arborii digitali (DT) înmagazinează șiruri de caractere cuvânt cu cuvânt. Fiecare nod conține un caracter. Fiecare nod are un fiu pentru fiecare literă a alfabetului (pentru limba engleză sunt de 26 litere, astfel încât avem 26 de fii). Aceasta ramificație introduce o mulțime de informații neutilizate, care sunt fiii nodurilor neutilizate. Pentru rădăcina, avem 20 noduri, care sunt goale. Rădăcina nu conține informații. Deci, dacă noi ar trebui să memorăm un DT pentru toate codificările Unicode Standard, care conțin 34000 de caractere, am ocupa 1 megabyte doar cu 1 nod, chiar dacă am folosi doar o pereche de fii. Căutarea în DT este din rădăcină jos, potrivește 1 caracter la un moment dat. Dacă suntem în căutarea unui cuvânt, de la rădăcinăvom merge la un nod, și de la acest nod am merge la n noduri. Căutarea este foarte rapidă.

#### Ternary Search Tree

TST (figura 4.9) se combină aceste funcționalități pentru a crea arborele de căutare final. Ca arborii, se descompune un cuvânt caracter cu caracter. Ca și BST-ul, TST-uleste eficient din punct de vedere al spațiului, deoarece un nod are trei fii, mult mai puțin decât dacă ar avea n.

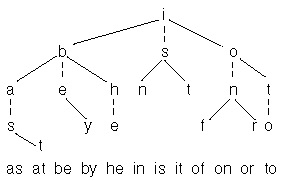


Figura 4.9 Ternary Search Tree

Fiul din stânga și fiul din dreapta păstreazăproprietatea arborelui binar de căutare, care afirmă că dacă cauți ceva și nodul curent deține o informație mai mare decât cea pe care o cauți, atunci mergi în stânga.Dacă informația din nod este mai mică decât ceea ce căutăm mergem în partea dreaptă. mai mică decât cea pe care il cautati, atunci te duci dreptate. Dar dacă este egal, mergi la fiul din mijloc, și continui cu următorul caracter din secvența de căutare. Un cuvânt este găsit atunci când corespund toate caracterele și nu sunteți pe un nod nul.

#### Tournament Tree si TST ”pe steroizi”

Pentru ca căutarea în arbore să fie una eficientă, fără explorarea arborelui ci mergerea fix la ținta căutată, Ternary Search Tree-ul împrumută o funcționalitate a unor arbori denumiți și Tournament Tree [[17](#Knu98)] [[18](#Hin01)] dar și metoda de memorare a unui Segment Tree din [[9](#Mat11)]. Un tournament tree simulează un arbore construit, de exemplu, la un campionat de fotbal, în care fiecare nod conține ”câștigătorul” subarborelui său.

Pentru intergrarea în contextul nostrul, fiecare nod din TST v-a conține weight-ul maxim din subarborii săi. Astfel, în momentul căutării, o să se verifice vecinii după weight (in cadrul parcurgerii BFS), iar în cazul în care aceștia respectă constrânderea de prefix, se merge mai departe. În acest fel, căutarea printr-un TST este exact la țintă și este una din cele mai rapide căutări posibile într-o astfel de structură.

#### Segment Tree

Segment Tree(ST) (figura 4.10), este un pic diferit. Se aseamănă un pic cu arborii B +, dar cu un arbore binar de căutarecopac segment structure. Sunteți interesat în căutarea de date de la o valoare care este în intervalul ce arborele de căutare îl ține.

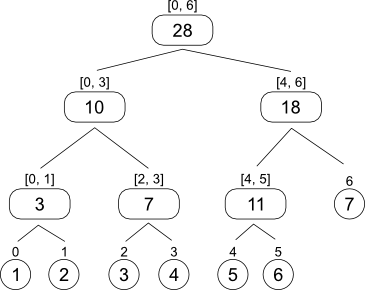


Figura 4.10 Segment Tree

Data ce o conține nodul din segment tree este un interval cu toate datele din subarborele nodului .

Fiul stâng conține informația de la limita stângă a intervalului de la mijloc al intervalului părintelui și fiul drept conține informația de la mijloc al intervalului părintelui de la limita dreaptă a intervalului părintelui. Trebuie să mergem pe această structură până la frunze (limita stânga este egal cu limita dreapta) pentru a găsi informații.

#### Combinarea arborilor și construcția căutării

Pentru un algoritm mai rapid de căutare, am combinat acești 2 arbori, TST si ST. Am folosit doar faptul că un segment tree memorează datele dintr-un interval, nu structura. Am șters toate straturile de la radacină la frunze, astfel încât rădăcina conține toate frunze. Practic puteți numi acest segment tree modificat o hartă hash, dar am rămas cu segment tree, deoarece cu acesta am început. Fiecare frunză are o referință la un arbore de căutare ternar, și fiecare arbore de căutare ternar conține cuvinte care încep cu aceeași literă.

Această schimbare este necesară în algoritm de când sistemul cunoaște exact la care frunză să meargă de fiecare dată, nu trebuie să mai parcurgă toată calea segment tree-ului.

Arborele de Ternary Search Tree v-a fi împărțit în arbori, fiecare arbore memorând toate cuvintele care încep cu o literă.

Această decizie a fost făcută din cauza problemelor ridicate de către căutarea cuvântul. Dacă sistemul a avut doar 1 arbore, nodurile au trebuit să conțină informații despre unde începe un cuvânt, unde se sfârșește el, care cuvânt începe acolo, care cuvânt se termină acolo, și viitorul arborelui dacă nu este utilizat cum trebuie. Dacă știm toate cuvintele ce încep de la rădăcină, trebuie doar să reconstituim traseul de la rădăcină la unii copii care au un atribut(flag) care spune dacă un cuvânt se termină acolo.

Un alt motiv este faptul că autocompletarea începe după ce prima literă este scrisă. Sistemul trebuie să caute numai cuvinte care au această literă prima.

Cel mai important motiv pentru această abordare, este atunci când un cuvânt este căutat, căile în arbore care nu se mulează pe prefixul de interogare sunt eliminate, ceea ce mărește viteza procesului de căutare

Abordarea noastră are un heap ca să memoreze dicționarul din considerențe de eficiență. Pentru ca cuvintele să fi în concordanță cu arborii de căutare, sistemul aplică aceeași divizare a cuvintelor pentru dicționar. Toate cuvintele care încep cu aceeași literă vor fi stocate în aceeași grămadă.

### Grafuri Orientate

Din descrierea din secțiunea 4.3.2, structura de date folosită pentru autocompletarea la nivel de frază este o funcție între două cuvinte dată de formula (4.7)

X și Y sunt cuvintele după care se face numărarea. Indicele i parcurge toate ngramele memorate în modelul de date, și contorizează de câte ori cuvântul y se află după cuvântul x în aceste ngrame. Pentru o mai bună eficiență, această funcție este întotdeauna precalculată, fiind doar actualizată în cazul apariției unei noi alăturări a perechii (x,y) în structuri de date adăugate ulterior.

Dacă analizăm mai departe această funcție, o structură de date care poate memora aceste informații este un graf orientat cu costuri.

**(Def)** Un graf orientat este o pereche de mulțimi *G=(V,A)* sau *G=(V,E)* , în care V reprezintă mulțimea nodurilor și E reprezintă mulțimea arcelor ce leagă aceste noduri. Arcele au proprietatea de direcție. Un arc este o relație de la un vârf v la un vârf u, notată *w(v,u)*, în care w este o funcție între aceste două noduri, iar valoarea funcției w este costul acestei relații. (Bang-Jensen and Gutin 2000)

Structura de graf orientat acoperă toate cerințele problemei noastre. Cuvintele vor fi asociate cu nodurile, iar funcția f(x,y) între aceste cuvinte v-a reprezenta un arc.

Structura pentru ngramele “ana are mere” și “maria are pere” este ilustrată în figura 4.11

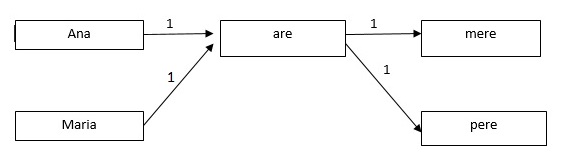


Figura 4.11

### Grafuri în loc de suffix trees

Pentru construcția autocompletării la nivel de frază se propune în [[17](#Nan071)] o structură de tip suffix tree (descrisă in capitolul 3). Această structură menține pentru fiecare cuvânt un arbore de tip suffix tree cu toate cuvintele ce au urmat vreodată în construcțiile ce au acest cuvânt ca început. Este o dulicare de memorie incredibil de mare.

În loc de această pădure imensă de arbori, soluția propusă în această lucrare concatenează toți acești arbori într-un graf iemns de dependințe între cuvinte. Weight-urile dintre cuvinte sunt date de suma frecvențelor apariției succesiuni respective de cuivne în toate ngramele. Această abordare are totuși un dezavantaj. Nu poate să facă distingerea între două sau mai multe ngrame ce trec prin același cuvânt, și poate unul este mai important ca celălalt. Pentru a compensa acest dezavantaj, reconstruirea ngramelor este făcută în așa fel încât să reflecte discrepanțele între 2 legături, una cu weight mic și una cu weight mare. Detalii în secțiunea următoare.

### Extragerea ngramelor din graf

Refacerea unui ngram dintr-un graf se face prin poziționarea pe primul cuvânt din ngram. Se parcurge graful în adâncime și se concatenează nodurile până când ngramul ajunge la dimensiunea dorită.

Weight-ul unui ngram este dat de media armonică a costurilor dintre cuvinte. Această medie simulează mai bine un weight al unui ngram decât media aritmetică și geometrică din cauza situației descrise în figura 4.12

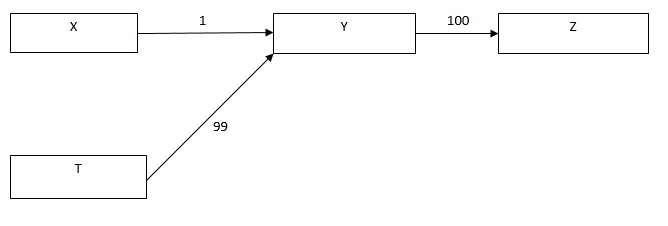


Figura 4.12

Această situație descrie memorarea a două ngrame, “x y z” ce a apărut o singură dată și “t y z” ce a apărut de 99 de ori. În mod normal, ngramul “x y z” ar trebui să aibe la recalculare un weight apropiat cu cel inițial.

În cazul mediei aritmetice, această valoare ar fi 50. În cazul mediei geometrice, această valoare ar fi 9.94. În cazul mediei armonice, această valoare ar fi 1.98. Media armonică returnează rezultatul cel mai apropiat de realitate, deci această metodă v-a fi folosită pentru refacerea weight-ului ngramelor.

## Filtrarea datelor

În cadrul acestei secțiuni, se va descrie conceptul de stop words si impactul acestora în autocompletare.

### Stop words

În secțiunea 4.3.1 am introdus cuvântul ca și element principal folosit de către soluția de autocompletare. Nu toate cuvintele pot fi folosite într-un sistem de autocompletare. Un pas de filtrare se va aplica atât la nivelul de date primite de către utilizator cât și la nivelul de date memorate în structurile de date.

După cum s-a descris în secțiunea 2.2, stop words nu sunt parte din datele ce sunt abordate în cadrul autocompletării. Acest lucru nu se aplică și la autocompletarea la nivel de frază.

În cadrul autocompletării a nivel de frază, stop words contează. Fără stop words, structurile de date nu sunt o înșiruire de cuvinte ce au sens, în orice limbă ar fi. Pentru că autocompletarea la nivel de frază introduce o secțiune întreagă de cuvinte. Pentru ca utilizatorul să nu fie nevoit să revină între cuvinte și să completeze aceste stop words, lucru ce ar face autocompletarea la nivel de frază să nu fie folosită niciodată întrucât încetinește viteza de scris, stop words nu sunt eliminate din aceste structuri.

### Filtrarea ngramelor și a cuvintelor

Filtrarea structurilor din autocompletarea la nivel de frază se face după semnele de punctuație. O virgulă delimitează două părți din propoziție care țin de contexte diferite, cuvintele despărțite de virgulă nefiind într-o relație unul cu altul. Punctul sau alte semne ce marchează sfârșitul unei propoziții delimitează și mai clar două contexte diferite.

Aceste lucruri sunt reprezentate grafic în figura 4.13.

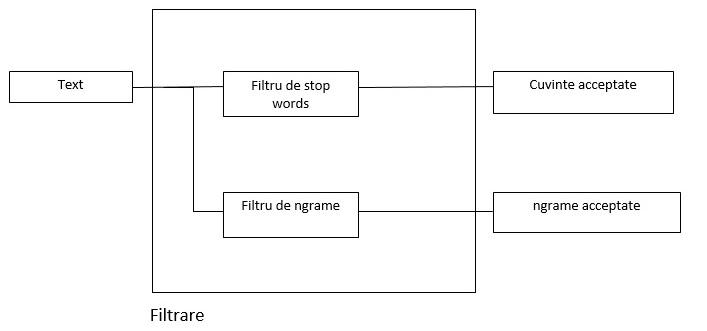


Figura 4.13. Filtrarea datelor

Filtrarea are două componente.

* Filtrul de stop words prin care textul este trecut pentru a da cuvintele ce vor fi memorate de structura de date pentru a fi folosite în autocopletarea de cuvinte.
* Filtrul de ngrame care extrage ngramele după regulile descrise mai sus, pentru a putea fi memorate și folosite în autocompletarea de frază.

Textul ce intră în aceste filtre este de două feluri:

* Text din textele folosite pentru construcția dicționarului implicit în etapa de învățare.
* Textul scris de utilizator în momentul folosirii soluției. În acest caz, textul va apărea caracter cu caracter pe măsură ce utilizatorul scrie.

Pentru că tratarea celor două tipuri de texte nu este uniformă, filtrele v-or avea două comportamente. Extragerea de date din texte lungi, sau construcția de date din texte ce vin caracter cu caracter.

## Arhitectura generală a mecanismului de autocompletare

Arhitectura solutiei de autocompletare este reprezentată în figura 4.14.

### Input - Output

*User input* sunt datele care vin de la utilizator. Aceste date pot fi de 8 tipuri:

Secvența scrisă de utilizator pentru care trebuie făcută autocompletarea la nivel de cuvânt (prefix de completare). Această dată este șirul de caractere ce a fost scris de utilizator. Dacă utilizatorul vrea să scrie cuvântul “acasă”, prin acest canal vor veni pe rând informațiile “a”, “ac”, “aca”, “acas”, etc. La fiecare tastă apăsată de utilizator se trimite ceea ce a scris el până atunci.

Cuvântul selectat de utilizator, indiferent dacă acesta a fost ales din lista de sugestii sau nu. Prin această informație se actualizează indicii de utilizator și de actualitate în cazul în care cuvântul exista deja în dicționarul de utilizator, sau se introduce un nou cuvânt cu acești indici.

Fiecare caracter scris. În acest mod se pot identifica ngramele care se scriu de către utilizator, pentru actualizarea grafului de cuvinte ce memorează ngramele.

Un fișier întreg de învățare. Acest input este folosit pentru crearea de dicționar implicit în cazul în care acesta nu există și utilizatorul dorește crearea unuia, în cazul în care utilizatorul dorește mărirea acestui dicționar, sau în cazul în care utilizatorul dorește modificarea acestui dicționar (suprascrierea).

Schimbarea de dicționar. Există posibilitatea existenței mai multor dicționare. Prin această comandă se specifică ce dicționar se încarcă.

Pornirea sistemului. Această comandă se declanșează la pornirea sistemului și se încarcă un dicționar în acest moment. Se construiește modelul de căutare din acest dicționar.

Salvarea dicționarului. Această comandă salvează starea dicționarului. Apelată în corelație cu apelul de schimbare de dicționar

Oprirea sistemului. Această comandă se declanșează la oprirea sistemului și are ca efect salvarea dicționarului curent și eliberarea memoriei de către sistem.

*Lista de sugestii de cuvinte* sunt datele care sistemul le returnează în cazul autocompletării de cuvinte. Aceste date, după cum zice lista, sunt returnate sub forma unei liste în care poziționarea elementelor se face după weight-ul acestora. Astfel, cuvintele vor fi sortate după weight. Cuvântul cu weight-ul cel mai mare apărând pe prima poziție în listă, iar cuvântul cu weight-ul cel mai mic apărând pe ultima poziție.

*Lista de sugestii de fraze* sunt datele care sistemul le returnează în cazul autocompletării de fraze. Aceste date sunt structurate la fel ca si *lista de sugestii de cuvinte*, doar că în loc de cuvinte sunt ngrame. Sortarea se face tot după weight-ul ngramelor.

*Persistența* este modului ce face citirea de pe disk și salvarea înapoi pe disk a dicționarelor. Modulul primește ca input-uri căile de acces către fișierele în care se află memorate aceste dicționare.

### Module ce țin structuri de date de tip căutare.

*Pădure de arbori de căutare* este modulul care gestionează și conține structura de date de căutare, arborii TST descriși în secțiunea 4.4.1.

*Graful de cuvinte* este modulul prezentat în secțiunea 4.4.2 ce memorează graful de autocompletare de frază. In acest modul sunt adăugate ngrame de către *filtrul de ngrame*.

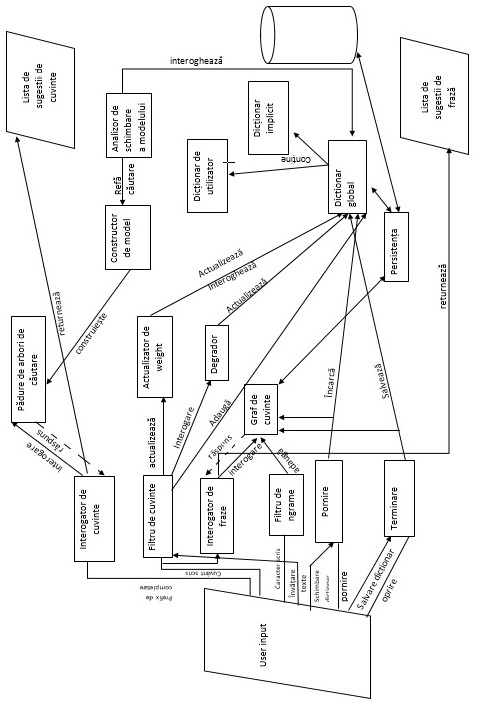


Figura 4.14. Arhitectura soluției de autocompletare

### Module ce țin structuri de date de tip indexare și stocare

*Dicționar implicit* este modulul ce reține dictionarul implicit. Aici se regăsesc toate perechile de cuvinte cu weight-urile lor implicite. Dicționarul primește cuvinte și date despre cuvinte (care este weight-ul implicit) de la *filtrul de cuvinte* în momentul în care sistemul primește de la utilizator comanda de învățare de texte. Funcționalitățile acestui dicționar sunt apelate de către modulul *dicționar global.*

*Dicționar de utilizator* este modulul ce conține dicționarul utilizatorului. Aici se regăsesc toate perechile de cuvinte cu weight-urile de utilizator și de actualitate. Dicționarul primește cuvinte și weight-urile de utilizator și de actualitate de la modulul *actualizator de weight*. Dicționarul primește aceste informații în momentul în care un cuvânt este folosit de utilizator și trece de filtru. Funcționalitățile acestui dicționar sunt apelate de către modulul *dicționar global.*

*Dicționar global* este modulul care înglobează cele două dicționare, cel implicit și cel de utilizator, și expune viziunea globală despre cuvintele din dicționar. Acest dicționar se mai numește, cum a fost referit în secțiunea 4.4.1, date de indexare. Aici se calculează weight-urile cuvintelor prin combinarea datelor din cele două dicționare. În momentul în care se cere o weight-ul unui cuvânt, se interoghează weight-ul implicit din *dicționar implicit*, weight-urile de utilizator și de actualitate din *dicționar de utilizator* și se combină după formula de calcul a weight-ului cuvântului (4.5). Dicționarul global are responsabilitatea de a salva cele două dicționare și de a incărca altele noi prin delegarea acestei sarcini modulului de *persistență*.

### Module ce realizează interogările.

*Interogator de cuvinte* este modulul care se ocupă de gestionarea autocompletărilor de cuvinte. Acesta primește ca date de intrare prefixul de completare, interoghează *pădurea de arbori de căutare*, primește completările pentru acea secvență și le returnează utilizatorului în lista de sugestii de cuvinte.

*Interogator de fraze* este modulul care se ocupă de gestionarea autocompletărilor de cuvinte. Acesta primește ca date de intrare cuvântul scris de utilizator, interoghează *graful de cuvinte*, primește completările pentru acel cuvânt și le returnează utilizatorului în lista de sugestii de cuvinte.

### Module ce filtrează datele de input

*Filtru de cuvinte* este modulul prezentat în secțiunea 4.5 ce realizează filtrarea de stop words și de cuvinte care au dimensiunea mai mică decât 4 (secțiunea 2.2). Primește ca input cuvântul scris de utilizator și delegă manipularea acestei operații *actualizatorului de weight* în cazul în care este un cuvânt valid. Acest filtru are rolul doar de delegator, nu face nici o manipulare a datelor.

*Filtru de ngrame* este modulul prezentat în secțiunea 4.5 ce realizează extragerea de ngrame după regulile descrise în aceeași secțiune. Modulul primește caracter cu caracter datele scrise de utilizator si trimite *grafului de cuvintei* ngramele identificate. În cazul în care primește un text întreg din textele de antrenare a dicționarului implicit, își va face extragerea datelor tot caracter cu caracter, folosind funcționalitatea ce se aplică în momentul în care se primește textul scris de utilizator.

### Module ce se ocupă cu construcția datelor de căutare

*Constructor de model* este modulul care preia datele din *dicționarul global* și construiește *pădurea de arbori de căutare*. După cum am expus în secțiunea 4.4, este modulul care construiește datele de căutare din datele de indexare. Modulul interoghează *dicționarul global*, primește de la acesta lista cuvintelor ordonate după weight și construielte în aceeași ordine fiecare arbore pentru fiecare literă din alfabet.

*Analizor de schimbare a modelului* este modulul care decide dacă în modelul de date un număr suficient de mare de cuvinte și-au schimbat weight-urile astfel încât căutarea să fie mai rapidă dacă arborele de căutare se reconstruiește. *Dicționarul global* are un submodul care memorează câte cuvinte și-au schimbat weight-ul și cât de mult. Acest submodul, numit *calculator de cost,* este interogabil și în funcție de un parametru configurabil, creează o funcție de cost ce evaluează această necesitate de refacere a modelului de date. *Calculatorul de cost* menține câte o funcție de cost pentru fiecare literă din alfabet, oglindind astfel structura aflată în *pădurea de arbori de căutare*. Dacă refacerea este necesară, analizorul trimite un semnal *constructorului de model* care reia acest proces de refacere a modelului de date de căutare, însă doar pentru arborele necesar. Constructorul nu reconstruiește toată pădurea de arbori, ci doar arborele pentru care este necesar acest lucru. Operația de refacere a modelului este transparentă utilizatorului, deoarece până când noul arbore este construit, interogarea se face tot pe vechiul arbore, iar la finalizarea construcției se schimbă doar pointerul arborilor, această operație realizându-se instantaneu. Separarea funcției de cost pe fiecare literă din alfabet are scopul de a scuti sistemul de calcule inutile și costisitoare, adică refacerea modelului de căutare pentru restul de n-1 arbori când arborele i necesită recalculare.

### Module ce se ocupă cu actualizarea weight-urilor cuvintelor

*Actualizatorul de weight* este modulul ce se ocupă cu actualizarea weight-urilor din *dicționarul de utilizator*. Modulul primește de la *filtrul de cuvinte* cuvintele folosite de utilizator. Modulul interoghează mai întâi dicționarul global pentru a vedea starea cuvintelor din modelul de date. Pe baza acestei interogări, setează weight-ul de actualitate și trimite la *dicționarul global* cuvântul împreună cu cele două weight-uri calculate.

*Degradorul* este modulul care se ocupă de calculul degradărilor weight-urilor de utilizator și de actualitate a cuvintelor folosite de utilizator. *Degradorul* primește ca input de la *Filtrul de cuvinte* un semnal reprezentând faptul că o completare s-a realizat. *Degradorul* contorizează acest lucru și interoghează fiecare cuvânt în parte, actualizând-ui aceste weight-uri. Având în vedere că la fiecare completare *Degradorul* trebuie să parcurgă toate cuvintele se poate spune că este o operație costisitoare, însă nu este așa. *Degradorul* parcurge doar cuvintele care fac parte din același arbore și doar în momentul în care utilizatorul alege o completare a sistemului. Din această cauză, frecvența acestei operații nu este foarte crescută.

### Module ce se ocupă cu inițializarea și distrugerea sistemului

*Pornire* este modulul care se apelează atunci când sistemul este initializat. Acest modul inițializează parametrii de mediu cu valorile citite din configurare și delegă operația de citire a dicționarului către *Dicționarul global*. De asemenea, reacționează și la operațiile de schimbare de dicționar întrucât operația de citire si scriere de dicționar este apelată din acest modul.

*Terminare* este modulul care se apelează atunci când mediul sistemului este oprit. Acest modul trimite operația de salvare a dicționarului și închide sistemul.

## Design Pattern-uri folosite în arhitectură

**(Def)** Un Pattern este o pereche problemă / soluție numită care poate fi aplicată în contexte noi, cu sfaturi despre cum se aplică în aceste noi situații și disctuțiile despre avantajele și dezavantajele acestei abordări.

### High Coesion and Low Coupling.

Aceste două design pattern-uri merg mână în mână. High Coesion se referă la faptul că o entitate trebuie să efectueze operații cât mai specializate și dependente una de alta. Low Coupling se referă la faptul că entitățiile trebuie să fie cât mai nelegate una de cealaltă, schimbarea uneia dintre entități să afecteze minimal entitățile ce depind de respectiva entitate.

Un element sau o entitate poate fi o clasă, un modul sau un subsistem.

Se observă corelația între cele două design-uri și din lista comună de beneficii.

#### High Coesion

Problema rezolvată de acest design pattern este cum să menținem complexitatea sistemului în niște parametrii de eficiență.

Soluția reprezintă asignarea de responsabilități astel încât coeziunea să rămână mare.

**(Def)** Coeziunea este o măsură a cât de legate între ele și focusate sunt responsabilitățile unui element. Un element cu responsabilități legate între ele și care nu trebuie să facă multe operații are coeziune mare.

Coeziunea mică nu este de dorit deoarece dă naștere la următoarele consecințe:

* Elementele sunt greu de înțeles
* Elementele sunt greu de reutilizat
* Elementele sunt greu de menținut
* Elementele sunt mereu afectate de schimbări

Beneficiile High Coesion sunt:

* Înțelegerea elementelor. Claritate și ușurința în învățare a elementelor
* Mentenanța elementelor. Îmbunătățirea și mentenanța elementelor este simplificată
* Low coupling este de obicei suportat
* Reutilizabilitatea elementelor. Module cu funcționalități reduse și strâns legate sunt ușor de refolosit.

#### Low Coupling

Problema rezolvată de acest design pattern este prin ce metode entitățile să fie independente una de cealaltă, impactul la schimbare să fie minimal și să crească refolosirea entităților.

Soluția reprezintă distribuirea responsabilităților între entități astfel încât cuplajul să fie cât mai mic posibil.

**(Def)** Cuplajul este o măsură a cât de conectat este un element de un alt element, câte cunoștiințe are despre acest element sau dacă depinde de implementările acelui element.

Intuitiv, low coupling înseamnă că un element nu este dependent de “prea multe alte elemente”.

Beneficiile Low Coupling sunt

* Mentenanța elementelor. Elementele sunt afectate puțin sau deloc de schimbări în alte elemente
* Înțelegerea elementelor. Este ușor de înțeles izolarea.
* Reutilizarea elementelor.

#### Exemple

Exemple de respectare a design pattern-urilor:

Împărțirea modulelor pe tipul de responsabilitate. După cum s-au descris și clasificat modulele (ex: module ce țin structuri de date de căutare, module ce țin structuri de date de indexare, module ce se ocupă de crearea structurilor de date de tip căutare, module ce se ocupă de actualizarea weight-urilor), acestea au fost create astfel încât fiecare să aibe o responsabilitate legată de o parte din ciclul de viață a sistemului. Creare, învățare, memorare, interogare, salvare și terminare. Orice altă responsabilitate ce depășește aceste cicluri de viață este delegată la alt modul care se ocupă cu acea operație. Astfel modulele care se ocupă de memorarea datelor de indexare (dicționarele), chiar dacă weight-urile cuvintelor acestor dicționare sunt actualizate foarte mult, dicționarele nu au cunoștiință despre regulile de actualizare a acestor weight-uri, ele expunând doar metode de actualizare către modulele responsabile cu această funcționalitate.

Legarea modulelor între ele prin conexiuni minimale. După cum reiese din descrierea și clasificarea modulelor, fiecare are cunoștiințe minimale despre restul sistemului. Fiecare modul are tangențe sau delegă operații la maxim 2 alte module. Spre exemplu, modulul de *interogare de cuvinte* vede din sistem doar faptul că primește un input, trebuie să interogheze o structură de date și să returneze un rezultat la utilizator. Modulul de *pădure de arbori de căutare* știe să țină cuvintele într-o structură de date foarte rapidă în căutare, să răspundă la interogări si să își primească datele de introdus în arbori de la un modul extern. Toate modulele din sistem sunt create în acest sens. Fiecare are o responsabilitate și un scop limitat ca întidere în arhitectura sistemului.

Scoaterea responsabilității de salvare a dicționarului pe disc din modulul *dicționar global* și introducerea modulului de persistență. Această separare respectă ambele design pattern-uri deoarece:

* Se scoate dependința de modalitatea de stocare externă din *dicționar*
* Se scoate o operație ce nu ține de dicționar și se pune în altă entitate dedicată acestui tip de operații.

### Controller

Design pattern-ul controler adresează problema de cine este responsabil pentru tratarea unui eveniment de input de utilizator

Solutia reprezintă asignarea responsabilității primirii sau manipulării unui eveniment unei clase care

* Reprezintă ca întreg sistemul, subsistemul, aparatul
* Reprezintă un caz de utilizare in cadru căruia evenimente apar.

**(Def)** Un eveniment de input de utilizator este un eveniment generat de un actor extern. Sunt asociante cu operații de sistem.

**(Def)** Un controller este un obiect ce nu este obiect de interfață utilizator și este responsabil pentru primirea sau manipularea unui eveniment de sistem. Un controller definește modalitatea de tratare a operației de sistem.

Exemple:

Toate modelele care primesc user input sunt create ca fiind entități de tip cotroller. *Interogatorul de cuvinte* și *interogatorul de fraze* nu pot fi altceva decât controllere. Aceste module primesc input din exteriorul sistemului (de la utilizator), delegă sarcinile către modelele ce fac căutare de completări și returnează rezultatele completării. Expun complet funcționalitatea așteptată de către sistem.

Modelele de *pornire* și *terminare* sunt simple controllere care se ocupă de delegarea operațiilor de salvare sau citire a dicționarelor, sau de încărcarea proprietăților sistemului.

*Filtrele* conțin puțină mai multă funcționalitate decât un simplu controller ca entitate întreagă, însă practic aceste modele nu sunt apelate în mod explicit de utilizator, sunt efecte colaterale funcționării văzute de către utilizator. Prin urmare, aceste modele nu ar trebui să fie în totalitate controllere.

### Composite

Design pattern-ul composite are ca problemă bătaia de cap în tratarea unui obiect și a componentelor acestuia în mod diferit pentru fiecare. Exemplul clasic este desenarea unei interfețe utilizator, a butoanelor, a tabelelor, a grid-urilor, și altele.

Soluția este tratarea uniformă a unui obiect și a compozițiilor acestui obiect uniform. Crearea de dependențe recursive, formând o structură arborescentă, ajută la ascunderea complexității acestor elemente.

Dicționarele sunt modelele ce se conformă pattern-ului de composite. În loc să trebuiască făcut update-ul pe fiecare dicționar separat, un cuvânt să trebuiască să fie adăugat în două sau mai multe dicționare de fiecare dată când este necesar, se apelează toate aceste funcții doar pe dicționarul global, acesta delegând la fiecare dicționar operația, fiecare dicționar acționând corespunzător.

### Strategy

În domeniul de progamare, pattern-ul strategy (cunoscut și ca policy) este un design software care permite selectarea comportamentului unui algoritm în momentul rulării, mai degrabă decât la compilare.

Pattern-ul:

* Definește o familie de algoritmi
* Încapsulează fiecare algortim
* Oferă posibilitatea de aschimba algoritmii înăutnrul acelei familii de algoritmi

Strategy lasă algoritmii să varieze independent de clientii care îl folosesc. Strategy este unul din pattern-urile incluse în toate cărțile de specialitate [[19](#Fre04)].

### Observer

Pattern-ul Observer este un pattern de design software în care un obiect, numit subiectul, menține o listă de dependenți, numiți observanți (eng. Observers), și îi notifică automat despre fiecare schimbare de stare, de obicei invocân do metodă.

Este în principiu folosit pentru implementarea manipulării distribuite de evenimtente. Acest pattern este de asemenea un factor cheie în arhitectura de tip model-view-controller.

## Flow-uri funcționale

Flow-urile fucționale ale aplicației sunt de fapt cazurile de utilizare ale aplicației.

Soluția, privită din exterior de către utilizator, realizează funcționalitățile descrise în figura 4.7

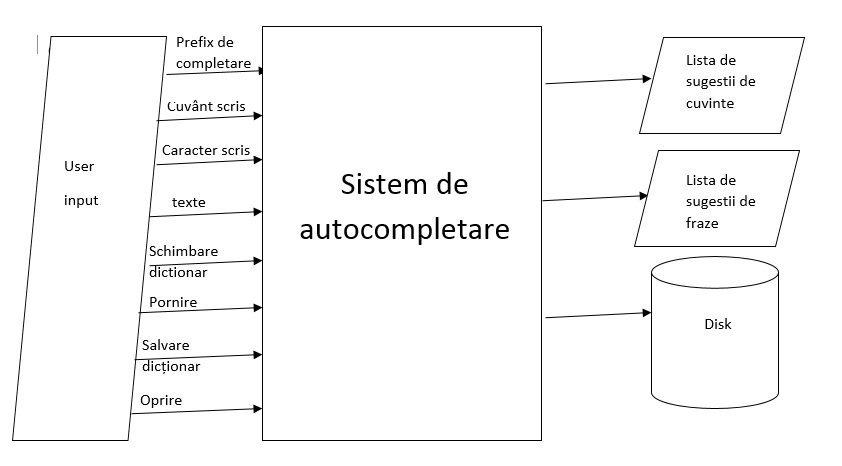


Figura 4.7. Sistemul de autocompletare ca black box

### Autocompletarea de text la nivel de cuvânt

Organigrama acestui flow este prezentată în figura 4.8.

În momentul în care aplicația este *pornită*, se *încarcă setările* și ultimele dicționare folosite. Utilizatorul poate alege să schimbe dicționarul sau să creeze altul nou, din fișierele lui.

Acești pași de initializare sunt prezentați deoarece sunt acțiuni necesare să se întâmple înaintea flow-ului din cazul de utilizare de autocompletare de cuvinte.

*Așteptarea de user input* este starea în care se află sistemul în momentul în care nici o interogare nu se află în desfășurare în sistem.

În momentul în care sistemul *primește un prefix* pentru autocompletare, acțiunea de *căutare a rezultatelor de autocompletare* se declanșează pe modulul de arbori de căutare. În urma acestei operații, se returnează rezultatele către utilizator.

In acest moment, fluxul de execuție poate merge în 3 direcții (cazurile 1, 2.a si 2.b), în funcție de acțiunile utilizatorului.

1. Dacă cuvântul pe care utilizatorul vrea să îl scrie se află în lista de sugestii. Utilizatorul va alege acest cuvânt, sistemul va actualiza modelul și se va întoarce în starea de așteptare de prefix de la utilizator
2. Dacă cuvântul pe care utilizatorul vrea să îl scrie nu se află în lista de sugestii. Acest caz are două subcazuri.
   1. Dacă ce a scris utilizatorul până acum nu este un cuvânt întreg. El v-a continua să scrie, iar sistemul v-a reveni la pasul de preluare de input de la utilizator.
   2. Dacă ce a scris utilizatorul până acum este cuvântul pe care el în dorește. Acest lucru denotă faptul că cuvântul nu este memorat în dicționar. În această situație, sistemul v-a adăuga cuvântul în dicționar, v-a actualiza modelul și va reveni la starea de așteptare de input de la utilizator

Modulele ce intră în realizarea acestui use case sunt: *Interogator de cuvinte*, *Pădure de arbori de căutare* pentru tratarea interogărilor utilizatorului. *Filtru de cuvinte, Actulizator de weight* pentru actualizarea modelului de date. *Dicționar global* pentru memorarea cuvintelor.

### Autocompletarea de text la nivel de frază

Când vine vorba de autocompletarea la nivel de frază, organigrama de funcționare este la fel cu cea de cuvânt.

În momentul în care aplicația este pornită, se încarcă setările și graful de completare. Autocompletarea de fraze nu are mai multe “dicționare”.

Așteptarea de user input este starea în care se află sistemul în momentul în care nici o interogare nu se află în desfășurare în sistem.

Momentul de declanșare a autocompletării (explicat în detaliu în secțiunea 4.2) în care utilizatorul scrie un cuvânt, acțiunea de căutare a rezultatelor de autocompletare se declanșează pe modulul ce se ocupă de manipularea grafului de ngrame. În urma acestei operații, se returnează rezultatele către utilizator.

Dacă utilizatorul alege una din aceste completări, atunci sistemul v-a actualiza ngramul respectiv în graf.

Modulele ce intră în realizarea acestui use case sunt: *interogator de fraze, graf de cuvinte* pentru tratarea interogărilor utilizatorului. *Filtru de ngrame* v-a primi ngramul selectat de utilizator (în cazul în care acesta alege o completare) și va delega *grafului de cuvinte* adăugarea ngramului.

### Invățarea de nou dicționar

Învățarea unui nou dicționar este o utilitate prin care un utilizator își selectează anumite fișiere din care să construiască un nou dicționar. Această funționalitate este utilă atunci când utilizatorul dorește să scrie într-un context cu totul nou față de ceea ce folosea până acum.

Un exemplu de diferențe între două contexte este următorul: Se consideră ca prim context scrierea de rețete de mâncare indiană. Acest context conține de multe ori cuvinte precum: “condiment”, “curry”, “miel”, “pui”, etc. Un al doilea context este cel de scrierea de rețete de mâncare italiană. Acest în cadrul acestui context găsim de asemenea multe din cuvintele folosite pentru mâncarea indiană, dar cu frecvențe mult mai mici. Cuvinte precum: “paste”, “spaghete”, “pizza” sunt mult mai folosite.

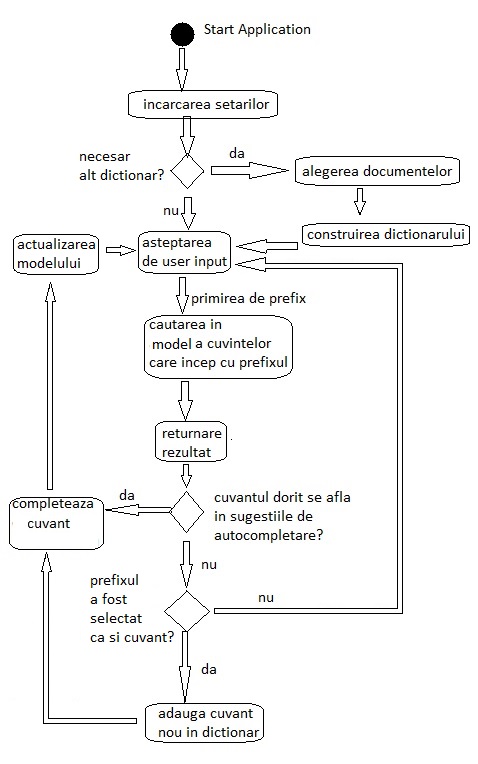


Figura 4.8. Organigrama funcțională a cazului de utilizare de autocompletare de cuvânt

Aceste două contexte sunt destul de asemănătoare și au ca si super-context un context comun, și anume contextul de rețete de mâncare. Însă ce se întâmplă când noi vrem să scriem un document științific despre autocompletare? În acest context nu există cuvinte precum “paste” sau “spaghete”, ci cuvinte precum “autocompletare”, “cuvânt”, “frecvență”, “sugestie”. În cazul în care utilizatorul dorește să scrie în acest context, un dicționar antrenat pe mâncare nu este folositor, iar până când el scrie suficient text pentru ca dicționarul de utilizator să umple aceste goluri, precizia sistemului este foarte scăzută.

Sistemul are nevoie de o astfel de funcționalitate pentru a fi mai mulat pe nevoile utilizatorului.

Organigrama acestui flow este prezentată în figura 4.9.

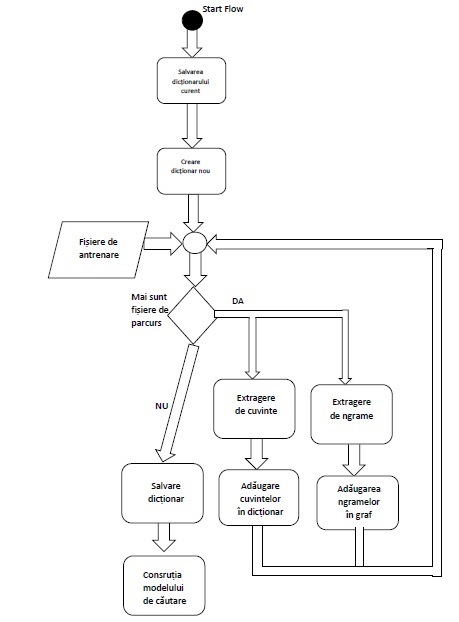


Figura 4.9. Organigrama flow-ului de învățare de nou dicționar.

### Adăugarea de noi fișiere la un dicționar existent

În ceea ce privește adăugarea de noi fișiere la un dicționar existent, organigrama v-a apărea fără etapele 2 și 3 de salvare a dicționarului curent și de creare a unui dicționar nou. Principiul rămâne același.

Pentru a realiza acest flow, este nevoie de 3 input-uri de la utilizator. Primul este input-ul de salvare de dicționar, primit de modulul *terminare*. Modulul spune dicționarului că este nevoie să fie salvat. Dicționarul delegă modulului *persistență* să salveze dicționarul curent.

Al doilea input este cel de schimbare de dicționar, care merge la modulul *pornire*. Modulul *Dicționar global* primește comanda și delegă modulul de *persistență* să creeze unul nou.

Dacă se dorește adăugarea de fișiere noi în dicționarul curent, primele două input-uri v-or fi omise.

Al treilea input este cel de învățare de texte. Sistemul primește fișierele ce conțin textele de învățat. *Filtru de cuvinte* și *filtru de ngrame* parcurg aceste texte și extrag informațiile, introducându-le în Dicționar și respectiv în graf.

După ce se termină extragerea informațiilor, se va reface modelul de căutare, iar apoi sistemul este gata să fie folosit cu noul dicționar.

# Proiectare de Detaliu si Implementare

În cadrul acestui capitol se v-or explica complet prin structura claselor componente și prin metodele de implementare a tuturor modulelor expuse în capitolul 4. De asemenea se v-a demonstra cum funcționează algoritmii de căutare după implementarea aleasă și de ce această modalitate de implementare este cea potrivită.

## Tehnologii

Modelarea arhitecturii a fost făcute ca să poată să fie implementată într-un limbaj de programare orientat obiect.

Limbajele de programare orientate obiect se clasifică în două mari categorii:

* Bazate pe prototipuri (Prototype-based)
* Bazate pe clase (Class-based)

Având în vedere că limbajele Prototype-based nu sunt studiate în facultate, nu o să aprofundăm acest domeniu.

Limbajele de programare orientat obiect demne de menționat pentru realizarea acestui proiect sunt:

* Java
* C#
* Python
* Ruby
* C++
* Groovy
* Smalltalk

Limbajul de programare ales pentru implementarea acestui sistem este Java. Din cauza lipsei de aplicații open-source sau de librării care realizează funcția de autocompletare într-un mod bun, scalabil, prietenos cu resursele și cu programatorul, am plănuit ca această aplicație să poată să fie construită ca o librărie, astfel încât să poată să fie importată în orice aplicație java, dând programatorului, cu câteva apeluri de metode, accesul la o soluție de autocompletare rapidă.

Java s-a ales și deoarece este limbajul cel mai studiat în Universitatea Tehnică din Cluj-Napoca. Din această cauză, orice student care dorește să continue acest proiect v-a avia o sarcină ușoară de a înțelege proiectul repede și de a extinde funcționalitățile lui

## Configurarea sistemului

Sistemul are atașat un fișier de configurare, numit autocompletionsystem.properties. În acest fișier se regăsesc perechi <cheie=valoare> ce descriu toți parametrii configurabili pentru acest sistem. Lista parametrilor configurabili sistemului este:

* file.directory.input = calea către directorul unde sistemul caută fișiere de intrare atunci când trebuie să creeze un dicționar nou sau să îl crească pe cel existent.
* dictionary.directory = calea către directorul unde sistemul menține dicționare.
* dictionary.name = numele fișierului care conține dicționarul pe care sistemul îl încarcă atunci când pornește.
* stop.words.name = numele fișierului care conține lista de stop words pe care sistemul trebuie să le filtreze
* system.alphabet = alfabetul pe care sistemul îl suportă. Sistemul v-a memora doar cuvintele care începu cu o literă din acest alfabet
* system.diacritics = (boolean, false sau true) această proprietate spune dacă sistemul v-a trata caracterele speciale separat sau le va converti la idemurile lor (”ă” la ”a”, ”î” la ”i”, ”ș” la ”s” ș.a.m.d).
* autocompletion.suggestions.size = cât de mare este lista pe care o întoarce sistemul în cazul ambelor autocompletări.
* autocompletion.suggestions.success.threshold = pragul din lista de sugestii de la care se consideră o sugestie acceptată.
* user.weight = cu cât crește frecvența weight-ului de utilizator la fiecare scriere a cuvântului de către utilizator. Acest parametru are scopul de a crește prioritatea unui cuvânt folosit de utilizator în cazul în care dicționarul implicit are weight-uri foarte mari pentru cuvintele din acesta, iar weight-ul cuvintelor folosite de utilizator o sa fie prea mic ca să influențeze cumva weight-ul total al cuvintelor. Implicit, weight-ul utilizator este incrementat cu maximul dintre 1 si (weight-ul cel mai mare din arborele cuvântului pe care îl actualizează sistemul) / 40
* weight.frequency.rate = cât din weight-ul total contează frecvența cuvântului din dicționarul implicit. Se introduce ca parte din 1 (ex: 0.3)
* weight.frequency.user.rate = cât din weight-ul total contează frecvența cuvântului din dicționarul utilizatorului. Se introduce ca parte din 1 (ex: 0.5)
* weight.actuality.user.rate = cât din weight-ul total contează actualitatea cuvântului din dicționarul utilizatorului. Se introduce ca parte din 1 (ex: 0.2)
* decay.alpha.actuality = alpha-ul funcției de degradare pentru weight-ul de actualitate
* decay.alpha.user = alpha-ul funcției de degradare pentru weight-ul de utilizator
* decay.queries.actuality = câte interogări asupra unui arbore contează cu o unitate în axa timpului în funția de degradare pentru weight-ul de actualitate
* decay.queries.user = câte interogări asupra unui arbore contează cu o unitate în axa timpului în funția de degradare pentru weigth-ul de cuvânt de utilizator

Motivul pentru care funcțiile sistemului sunt configurabile este simplu. Acest sistem v-a fi discponibil de integrat în orice altă aplicație java, iar dacă nu se poate configura, anumiți utilizatori a acestei librări nu v-or putea folosi la maxim acest sistem.

## Funcționarea algoritmilor de căutare a completărilor

În această secțiune se va explica funcționarea algoritmilor de căutare a sugestiilor de cuvinte cât și a sugestiilor de frază.

### Completarea de cuvânt

Figura 5.1 evidențiază arborele construit din dicționarul ce conține cuvintele: ”acea=2”, ”aceasta=21”, ”acasa=3”, ”acest=13”, ”aceste=6”, ”acestea=6”, ”acestei=4”, ”acele=5”, ș.a.

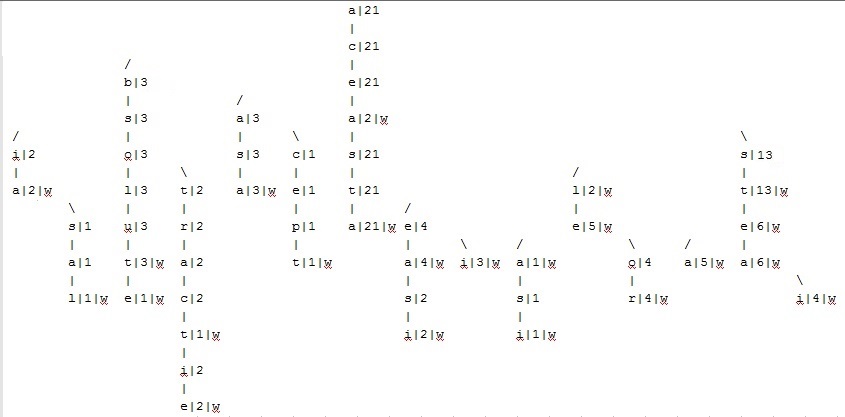


Figura 5.1. Arbore de căutare.

Algoritmul de căutare de cuvinte folosește o structura de max heap pentru memorarea căilor parcurse din arbore. Heapul tine minte nodurile de unde mai trebuie parcurs arborele la fiecare pas.

Parcurgerea se apropie de comportamentul de Breath First Search (BFS), doar că nodurile de parcurs nu sunt puse într-o coadă și scoase în ordinea introdusă, ci sunt puse sortate după weight-ul maxim aflat în subarborele nodului respectiv. Pentru a avea o structură sortată în care operațiile de ștergere și inserare sunt frecvente, iar structura trebuie să fie performantă, s-a ales folosirea unui MaxHeap.

Un heap este un arbore ce are proprietarea că fiecare nod este mai mare decât ambii copii. Inserarea și ștergerea se fac în operații de O(log h), unde h este înălțimea arborelui. Un heap este un arbore echilibrat, așadar complexitatea operațiilor este întotdeauna aceeași, nu poate tinde spre O(n), unde n este numărul de noduri din heap.

Pe lângă heap se mai reține o listă cu cuvinte identificate până la pasul curent, și o listă de cuvinte cu completări găsite. Lista de cuvinte identificate până la pasul curent (să o numim cuvinteGăsite) este necesară deoarece cuvinte cu weight mic pot apărea la începutul arborelui. Exemplu se vede și în figura 5.1 în cazul cuvântului ”acea=2”. Chiar dacă are weight mic, el se regăsește în primele cuvinte deoarece se suprapune peste cuvântul ”aceasta=21”. Lista de cuvinte cu completări găsite (să o numim completări) conține cuvintele cu cel mai mare weight care v-or fi returnate ca și listă de completări. Un cuvânt trece din lista cuvinteGăsite în lista completări în momentul în care weight-ul nodului din vârful heap-ului este mai mic decât al cuvântului din lista cuvinteGăsite. Acest lucru se întâmplă deoarece nu se mai poate găsi un cuvânt cu weight mai mare în arbore decât al cuvântului din lista cuvinteGăsite.

Pentru exemplul, un utilizator tastează ”ace” ca input. Dimensiunea sugestiilor de completare este luată ca 5.

Trasarea algoritmului se regăsește în tabelul 5.1

|  |  |  |  |
| --- | --- | --- | --- |
| **Nod parcurs** | **Heap** | **cuvinteGăsite** | **Completări** |
| Start | a=21 |  |  |
| a=21 | c=21 |  |  |
| c=21 | e=21, b=3 |  |  |
| e=21 | a=21, s=13, a=3, b=3 |  |  |
| a=21 | s=21, s=13, a=3, b=3 | acea=2 |  |
| s=21 | t=21, s=13, a=3, b=3 | acea=2 |  |
| t=21 | a=21, s=13, a=3, b=3 | acea=2 |  |
| a=21 | s=13, a=3, b=3 | aceasta=21, acea=2 |  |
| s=13 | t=13, l=5, a=3, b=3 | acea=2 | aceasta=21 |
| t=13 | e=6, l=5, a=3, b=3 | acest=13, acea=2 | aceasta=21 |
| e=6 | a=6, a=5, l=5, a=3, b=3 | aceste=6, acea=2 | aceasta=21, acest=13 |
| a=6 | a=5, l=5, i=4, a=3, b=3 | acestea=6, acea=2 | aceasta=21, acest=13, aceste=6 |
| l=5 | a=5, e=5, i=4, e=4, a=3, b=3 | acel=2, acea=2 | aceasta=21, acest=13, aceste=6, acestea=6 |
| a=5 | e=5, i=4, e=4, a=3, b=3 | acesta=5, acel=2, acea=2 | aceasta=21, acest=13, aceste=6, acestea=6 |
| sfarsit | e=5, i=4, e=4, a=3, b=3 | acel=2, acea=2 | aceasta=21, acest=13, aceste=6, acestea=6, acesta=5 |

Tabel 5.1. trasarea algorimtului de căutare a completărilor de cuvinte

După parcurgerea căutării, arborele a fost parcurs după cum arată figura 5.2

În cazul în care utilizatorul alege să nu folosească o completare din cele sugerate (el căutând să scrie cuvântul acela), utilizatorul v-a continua să scrie litera l. Sistemul știe să determine dacă interogarea de completare este nouă sau este continuată. Dacă interogarea completarea este nouă, starea se resetează și căutarea se face de la începutul arborelui. Altfel, căutarea continuă de unde a rămas sistemul, cu observația că listele cuvinteGăsite și completări se curăță doar de cuvintele care nu se mai potrivesc peste prefixul nou introdus.

Trasarea continuării algoritmului pentru prefixul ”acel” se regăsește în tabelul 5.2

Se observă faptul că în lista de cuvinteGăsite v-a rămâne cuvântul acel, iar în lista de completări nu va rămâne nici un cuvânt.

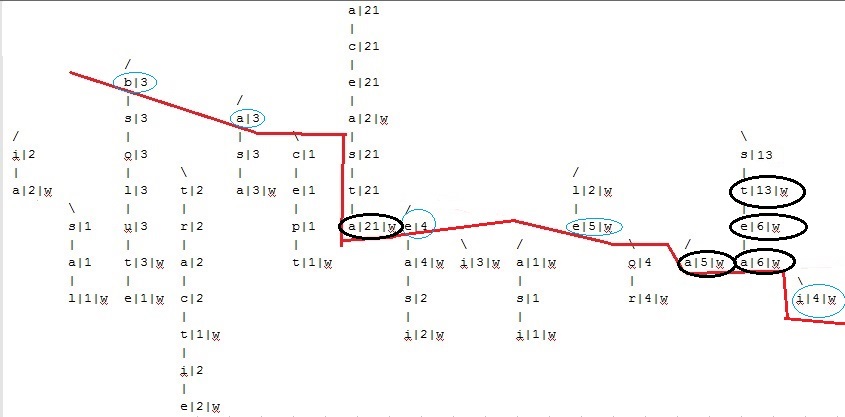


Figura 5.2. Parcurgerea arborelui la finalul interogării.

|  |  |  |  |
| --- | --- | --- | --- |
| **Nod parcurs** | **Heap** | **cuvinteGăsite** | **Completări** |
| start | e=5, i=4, e=4, a=3, b=3 | acel=2 |  |
| e=5 | i=4, e=4, o=4, a=3, b=3, a=1 | acel=2, acele=5 |  |
| i=4 | e=4, o=4, a=3, b=3, a=1 | acel=2 | acele=5 |
| e=4 | o=4, a=3, b=3, a=1 | acel=2 | acele=5 |
| o=4 | r=4, a=3, b=3, a=1 | acel=2 | acele=5 |
| r=4 | a=3, b=3, a=1 | acel=2, acelor=4 | acele=5 |
| a=3 | b=3, a=1 | acel=2 | acele=5, acelor=4 |
| b=3 | a=1 | acel=2 | acele=5, acelor=4 |
| a=1 | s=1 | acela=1 | acele=5, acelor=4, acel=2 |
| s=1 | i=1 |  | acele=5, acelor=4, acel=2, acela=1 |
| i=1 |  | acelasi=1 | acele=5, acelor=4, acel=2, acela=1 |
| terminare |  |  | acele=5, acelor=4, acel=2, acela=1, acelasi=1 |

Tabel 5.2. Continuarea trasării algoritmului pentru o interogare continuă

Arborele trasat în acest moment arată ca în figura 5.3. Se observă că pe ramurile ”acb”, chiar dacă erau weight-uri mai mari, nu s-a intrat deoarece nu se v-or găsi cuvinte cu același prefix. Această proprietate este dată de structura de Ternary Search Tree.

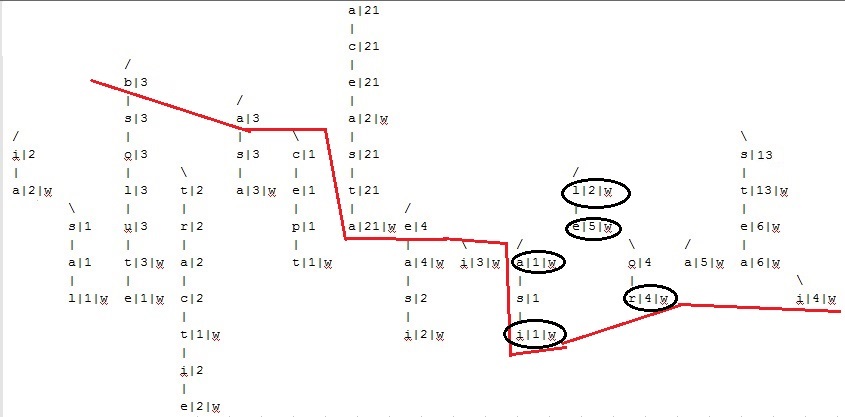


Figura 5.3. Arborele la finalizarea parcurgerii

### Completarea de frază.

Figura 5.4 evidențiază un graf cu dependințe între cuvintele x, y, z și t.

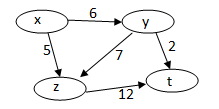


Figura 5.4. Graf de căutare.

Ngramele incluse în acest graf sunt ”x y z=4”, ”x y t =2”,”x z t=5” și ”y z t=3”.

Algoritmul presupune construcția tuturor ngramelor ce încep din x și au adâncime 3, adică toate tri-gramele. Identificarea trigramelor se face cu algoritmul de Breadth First Search. După identificarea trigramelor, se calculează weight-ul acestora și se introduc într-un maxheap. Se returnează primele k ngrame.

Algoritmul are 2 structuri de date. Coada folosită pentru parcurgerea in lățime și heapul de sortare a ngramelor calculate.

Trasarea algoritmului de sortare se face în tabelul 5.3

|  |  |
| --- | --- |
| **Nod parcurs** | **Coadă** |
| start | [] |
| x | [y={6;”x y”}, z={5;”x z”}] |
| y | [z={5;”x z”}, z={6,7;”x y z”}, t={6,2;”x y t”}] |
| z | [z={6,7;”x y z”}, t={6,2;”x y t”}, t={5,12;”x z t”}] |
| end |  |

Tabel 5.3 Trasarea algoritmului de căutare a ngramelor

Algoritmul se termină atunci când primul element din coadă este un nod identificat la o distanță de 2 față de nodul de start.

Pentru calcularea weight-urilor ngramelor se ia coada obținută acum, de la cap la coadă, și pentru fiecare ngram se aplică formula de calcul (media armonică a weight-urilor conexiunilor dintre cuvinte).

Rezultatele calculării weight-urilor ngramelor sunt următoarele: ”x y z=6.46”, ”x y t=3”, ”x z t=7”. Se observă că ordinea ngramelor rămâne aceeași după extragere ca și înainte de introducerea în arbore.

## Implementarea modulelor sistemului

### Input - Output

Datele care vin de la utilizator sunt reprezentate în diagrama arhitecturii proiectului (figura4.6) ca fiind săgețile ce pleacă din căsuța numită user input. Datele ce vin prin aceste puncte sunt:

Secvența scrisă de utilizator pentru care trebuie făcută autocompletarea la nivel de cuvânt (prefix de completare). Acest input este asociat cu tipul de date String. Cuvântul selectat de utilizator este tot un String.

Fiecare caracter scris, primit pentru mecanismul de autocopletare de fraze, este reprezentat prin tipul de date Character.

Inputul ce marchează fișierele de învățare este defapt un semnal pentru sistem. Calea până la folder-ul în care se află fișierele ce urmează a fi citite se află în fișierul de proprietăți. Sistemul v-a căuta folderul și v-a parcurge fiecare fișier din acest folder. Este necesar ca fișierele să fie documente cu extensia ”.txt” sau să conțină doar text.

Semnalul de schimbare de dicționar este defapt numele fișierului în care este memorat dicționarul ce va fi încărcat în sistem.

Pornirea sistemului. Această comandă este implicită pentru sistem, acesta inițializându-se în urma citirii fișierului de configurare. Este necesară instanțierea clasei principale și atât.

Salvarea dicționarului. Această comandă este un semnal pentru sistem. Se salvează în fișierul de unde s-a citit dicționarul curent.

Oprirea sistemului reprezintă tot un semnal dat sistemului. Sistemul v-a salva dicționarul curent și v-a asigna toți pointeri la null pentru a declanșa garbage colletion și a elibera memoria folosită de sistem

*Lista de sugestii de cuvinte* este returnată ca un List<String>, adică o listă de Stringuri. Ordinea sugestiilor este dată de poziționarea în listă. Astfel, elementul de pe poziția 0 va avea weight-ul cel mai mare.

*Lista de sugestii de fraze* este returnată tot ca un List<String>. Nu s-a folosit o altă clasă pentru returnarea ngramelor deoarece clientul nu are nevoie să știe cum se comportă un ngram, ce înseamă sau cum trebuie să îl manipuleze. Are nevoie doar de reprezentarea vizuală a acestuia.

*Persistența* este modulul ce face citirea de pe disk și salvarea înapoi pe disk a dicționarelor. Modulul este construit ca în figura 5.5.

Clasa CharacterFilter verifică cuvinte. În cazul în care, din configurare, system.diacritics este setat la fals, atunci modifică toate caracterele speciale din cuvânt. Returnează cuvântul modificat.

Clasa DicționaryProcessor are responsabilitatea de a citi și de a salva dicționare pe și de pe disc. Ține minte directorul cu dicționare (setat în configurare) și numele dicționarului curent memorat. La salvare primește dicționarul ce urmează să fie salvat, și la citire primește numele dicționarului de citi, sau cel din configurare dacă nu se specifică.

Clasa FilesProcessor are responsabilitatea de a crea dicționare noi sau de a adăuga noi fișiere dicționarelor existente. Clasa depinde de DictionaryProcessor în cazul în care trebuie adăugate fișiere dicționarelor existente pentru a salva dicționarul modificat. Salvarea dicționarului este responsabilitatea DictionaryProcessor.

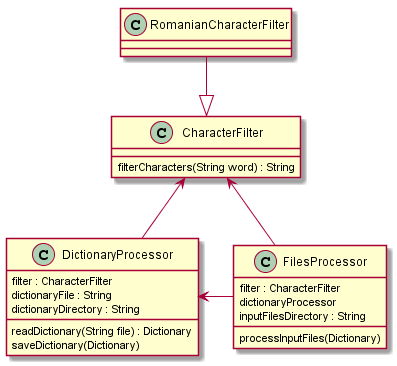


Figura 5.5. Componența modulului *Persistență*

### Module ce țin structuri de date de tip căutare.

*Pădure de arbori de căutare* este modulul care gestionează și conține structura de date de căutare, arborii TST descriși în secțiunea 4.4.1. Implementarea acestui modul arată ca în figura 5.6.

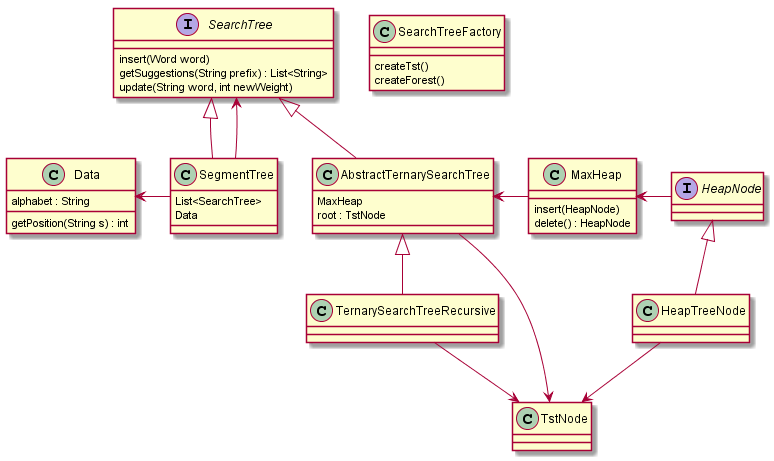


Figura 5.6. Pădurea de arbori de căutare.

Interfața SearchTree este interfața comună tuturor arborilor de căutare. Aborii de căutare, TernarySearchTree și SegmentTree implementează această interfață.

Se folosește pattern-ul Composite în acest modul. Având în vedere că avem mulți arbori de căutare, câte unul pentru fiecare literă din alfabet, avem nevoie de o modalitate de manipulare a acestora opacă pentru codul apeland si decuplată de implementare. Probabil în viitor vor fi mai mulți arbori, grupați după grupuri de litere din alfabet. Dacă așa e cazul, sistemul v-a fi supus unor mari schimbări arhitecturale.

Compoziția din acest modul se realizează făcând un arbore de tip segment tree peste arborii de căutare, iar acest arbore segment să fie văzut și el ca un arbore de căutare. Prin structura de moșteniri se respectă acest lucru. Restul sistemului vede doar Interfața de SearchTree. SegmentTree delegă orice funcționalitate subarborilor de căutare. Toate condițiile pattern-ului sunt îndeplinite.

Pentru crearea de arbori se folosește un Factory. Această clasă, SearchTreeFactory implementează desgin pattern-ul factory method, care spune că o interfață nu ar trebui instanțiată direct în cazul în care există mai multe implementări (SearchTree) ci ar trebui împățită această funcționalitate unei alte clase de tip factory, ce expune metode de construcție a acestei interfețe. Această ascundere a construcției ajută decuplarea codului și ușurința de schimbare a implementărilor claselor.

Arborele SegmentTree implementează conceptul de pădure. Acest arbore are ca frunze alți arbori, și realizează delegarea operațiilor acelui arbore care trebuie să trateze acea operație. SegmentTree ține acești arbori de căutare nu într-o structură de date arborescentă, ci într-o listă. Această decizie de implementare a fost luată pentru că de fiecare dată când trebuia să se acceseze arborele 5, trebuia să se parcurgă toată structura arborescentă până la frunze. Fiecare accesare însemna O(log n) timp pierdut (n = dimensiunea alfabetului). Cu ajutorul listei, timpul de acces este O(1) în cazul în care poziția în listă se cunoaște. De interogarea poziției se ocupă clasa Data.

Heapul este folosit cum este descris în secțiunea 5.3.1 pentru algoritmul de găsire a completărilor.

*Graful de cuvinte* este modulul prezentat în secțiunea 4.4.2 ce memorează graful de autocompletare de frază. Modulul este construit precum în figura 5.7

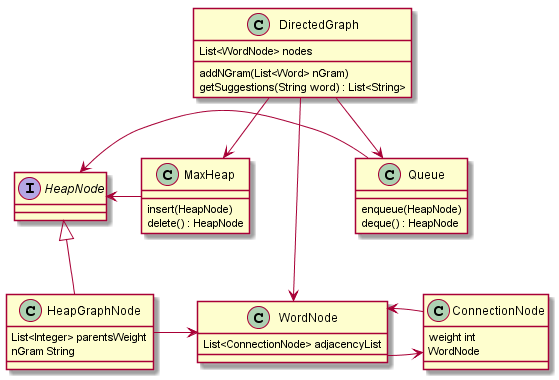


Figura 5.7. Graful de cuvinte

Graful este reprezentat sub forma listelor de adiacență. Graful memorează lista de cuvinte din ngrame. Cuvintele sunt nodurile arborelui. Fiecare cuvânt memorează lista de adiacență a lui. Lista de adiacență reprezintă fiecare cuvânt care urmează după cuvântul respectinv într-un ngram. Pentru lista de adiacență este necesară o clasă intermediară, ConnectionNode, care menține nodurile din graf. Conexiunea și weight-ul pentru fiecare arc.

Clasa DirectedGraph reprezintă implementarea grafului orientat. Graful are cuvinte memorate prin clasa WordNode, ce are o listă de adiacență și cuvântul pe care în memorează. Pentru construcția ngramelor se folosește clasa HeapGraphNode. Această clasă ține minte weight-urile arcelor prin care trece parcurgerea în adâncime și ngramul construit din concatenarea cuvintelor.

Clasa ConnectionNode reprezintă un arc. Ține minte weight-ul arcului și nodul în care merge conexiunea. Celălalt capăt este dat de nodul ce conține obiectul de tip ConnectionNode.

### Module ce țin structuri de date de tip indexare și stocare

*Dicționar implicit* este modulul ce reține dictionarul implicit. Aici se regăsesc toate perechile de cuvinte cu weight-urile lor implicite. *Dicționar de utilizator* este modulul ce conține dicționarul utilizatorului. Aici se regăsesc toate perechile de cuvinte cu weight-urile de utilizator și de actualitate. *Dicționar global* este modulul care înglobează cele două dicționare, cel implicit și cel de utilizator, și expune viziunea globală despre cuvintele din dicționar. Aici se calculează weight-urile cuvintelor prin combinarea datelor din cele două dicționare.

Clasele de dicționar memorează cuvintele componente ale dicționarului. Superclasa Dictionary oferă comportament comun pentru cele două dicționare. De asemenea are o listă de dicționare componente. Se creează astfel un composit, dicționarul global delegând operațiile către toate subdicționarele, combinând rezultatele pentru vederea globală a lucrurilor. Exemplu este operația de calculate a unui weight. În momentul în care se cere o weight-ul unui cuvânt, se interoghează weight-ul implicit din *dicționar implicit*, weight-urile de utilizator și de actualitate din *dicționar de utilizator* și se combină după formula de calcul a weight-ului cuvântului (4.5).

După cum sunt descrise modulele, diagrama claselor acestora ar trebui să arate ca în figura 5.8

O mare problemă a acestui design este faptul că orice interogare și orice manipulare a cuvintelor trebuie făcută pe toate sub-dicționare componente a dicționarului global. Acest lucru introduce multă extra-computație unde nu ar fi nevoie de acest lucru.

Rezolvarea elegantă a acestei probleme este mutarea structurii de dicționar din clasele de dicționar efectiv în clasele ce cuvinte. Acest lucru se vede în diferența dintre diagramele din figurile 5.8 și 5.9. Dicționarele implicit și de utilizator diferă unul de altul doar prin weight-urile ce le memorează pentru cuvintele respective, sau dacă un cuvânt există într-un dicționar dar nu există în celălalt. Dacă concatenăm cele două dicționare într-un dicționar global, și împărțim cuvintele și weight-urile acestora, rezolvăm problema accesări de două ori a dicționarelor pentru fiecare operație.

Actualizarea weight-urilor se face prin delegarea acestei operații unor structuri de date dedicate acestei operații. Prin această delegare, se face separarea regulilor de actualizare față de structura de cuvânt, astfel încât adăugarea de noi reguli sau ștergerea acestora să se realizeze fără a modifica structurile de cuvinte, ci doar structurile ce rețin regulile de actualizare. Aceste structuri sunt exemplificate prin interfata WeightUpdate și implementările pentru dicționarul default, DefaultWeightUpdate, și pentru dicționarul utilizator, UserWeightUpdate. Pentru a separa și mai mult structura de cuvânt de implementarea regulilor de actualizare, crearea acestor clase se face printr-un factory. Structura de word astfel are referință doar spre acest factory și interfața WeightUpdate. Această structură de clase implementată la actualizarea weight-urilor se numeste Strategy design pattern.

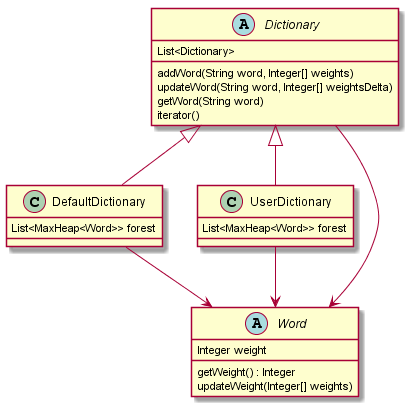


Figura 5.8. Diagrama claselor de dictionar 1

### Module ce realizează interogările.

*Interogator de cuvinte* primește ca date de intrare prefixul de completare, interoghează *structura de căutare*, primește completările pentru acea secvență și le returnează utilizatorului în lista de sugestii de cuvinte.

*Interogator de fraze* primește ca date de intrare cuvântul scris de utilizator, interoghează *graful de cuvinte*, primește completările pentru acel cuvânt și le returnează utilizatorului în lista de sugestii de cuvinte.

Aceste module sunt alcătuite din câte o clasă fiecare, exemplificate în figura 5.10 și 5.11.

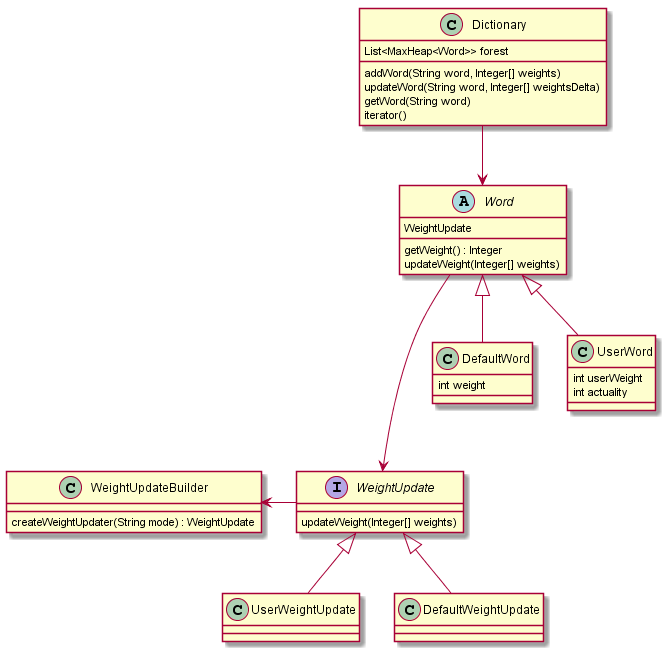


Figura 5.9. Diagrama claselor de dictionar 2

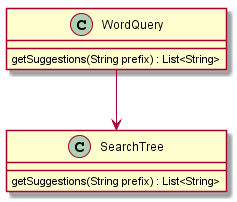


Figura 5.10. Interogator de cuvânt

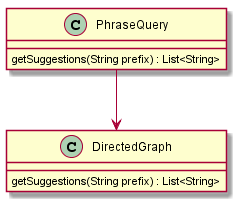


Figura 5.11. Interogator de frază

### Module ce filtrează datele de input

*Filtru de cuvinte* realizează filtrarea de stop words și de cuvinte care au dimensiunea mai mică decât 4 (secțiunea 2.2). Primește ca input cuvântul scris de utilizator. În cazul în care cuvântul trece de validatori, atunci flow-ul continuă la actualizatorul de weight. Componența claselor se poate observa în figura 5.12.

Se aplică același design pattern de strategy în ceea ce privește validatorii, cum s-a aplicat la clasele de update de user. Factory pentru construcția lor, Filter ca interfață, StopVordsFilter pentru filtrarea de stop words și LengthFilter pentr filtrarea de cuvinte scurte. Adăugarea, modificarea sau eliminarea de filtre este foarte simplă, codul manipulator nefiind schimbat deloc, doar codul ce construiește filtrele (adăugăm linia ce inițializează filtrul adăugat).

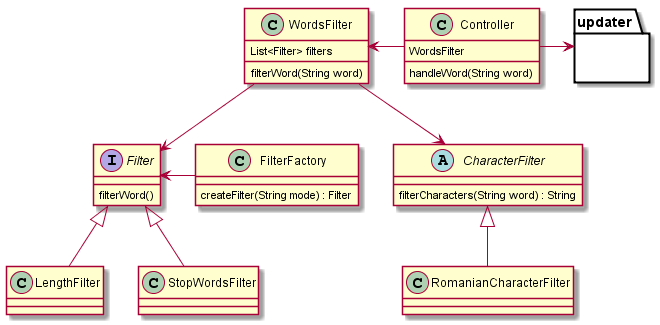


Figura 5.12. Filtrarea de cuvinte

*Filtru de ngrame* realizează extragerea de ngrame. Modulul primește caracter cu caracter datele scrise de utilizator si trimite *grafului de cuvintei* ngramele identificate. În cazul în care primește un text întreg din textele de antrenare a dicționarului implicit, își va face extragerea datelor tot caracter cu caracter, folosind funcționalitatea ce se aplică în momentul în care se primește textul scris de utilizator.

Filtrul este mai mult un automat de stare. Acest automat are următoarele stări:

* Primește literă. Concatenează la cuvântul curent
* Primește spațiu alb. Cuvântul anterior s-a terminat de completat. Dacă caracterul anterior a fost tot spațiu alb, ignoră. Verifică existența unui ngram. Dacă există, trimite la graf ngramul spre memorare, și începe ngram nou.
* Dacă este semn de punctuație, resetează starea automatului.

Clasele ce intră în realizarea acestui modul sunt prezentate în figura 5.13.

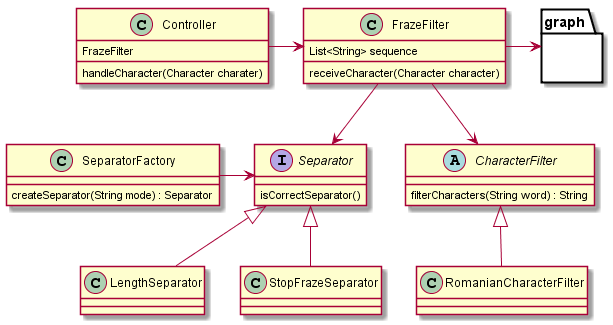


Figura 5.13. Filtrul de fraze

Filtrul de fraze are nevoie de filtrarea diacriticelor (dacă acestea nu sunt suportate). Se respectă modul de creare a separatoarelor, dar de data aceasta implementarea filtrului depinde de separatoarele ce există, deoarece în funcție de rezultatul acestor condiții, filtrul v-a trimite ngramul la graf sau va reseta căutarea de ngrame.

### Module ce se ocupă cu construcția datelor de căutare

*Constructor de model* este modulul care preia datele din *dicționarul global* și construiește *pădurea de arbori de căutare*, adică construiește datele de căutare din datele de indexare. Modulul interoghează *dicționarul global*, primește de la acesta lista cuvintelor ordonate după weight și construielte în aceeași ordine fiecare arbore pentru fiecare literă din alfabet.

Acest modul este alcătuit dintr-o singură clasă, numită TreeBuilder. Această clasă are metoda constructSearchModel care v-a construi modelul de căutare.

Schema clasei este prezentată în figura 5.14

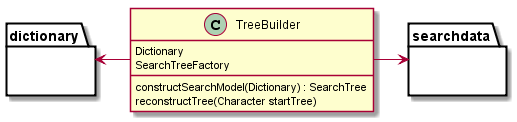


Figura 5.14. Constructorul de Model

*Analizor de schimbare a modelului* contorizează schimbările efectuate în dicționar în ceea ce privește ordonarea cuvintelor după weight. După cum s-a descris în secțiunea 4.4.2, dicționarul este memorat printr-un heap iar metoda de determinare dacă este nevoie de refacerea modelului este descrisă în secțiunea 4.4.3.

*Dicționarul global* are un submodul care memorează câte cuvinte și-au schimbat weight-ul și cât de mult. Acest submodul este implementat folosind design pattern-ul Observer. Designul este aplicat pentru construcția *Calculatorului de cost*. Heap-ul este acela care declanșează o acțiune de actualizare a unei funții de cost, deci heapul este un Observable. Funcția de cost tratează aceste acțiuni de actualizare a valorii, deci Calculatorul de Cost v-a fi Observer-ul.

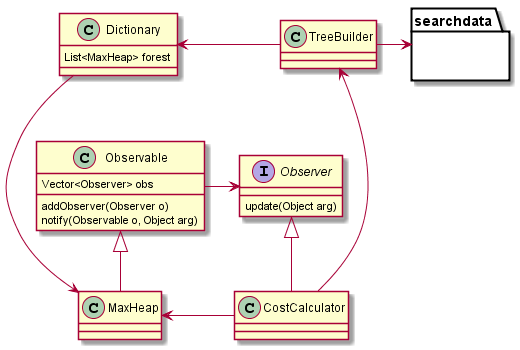


Figura 5.15. Analizarea schimbării modelului

### Module ce se ocupă cu actualizarea weight-urilor cuvintelor

*Actualizatorul de weight* conține regulile de actualizare a weight-urilor din *dicționarul de utilizator*. Tot modulul este alcătuit din clasa Updater, care conține regulile. O regulă de actualizare este o simplă formulă matematică. De exemplu, pentru actualizarea weight-ului de actualitate, valuarea cu care se incrementează este 1/r din weight-ul parțial al cuvântului din vârful arborelui din care face parte cuvântul pentru care se actualizează acest weight. R este parametrul care arată de câte ori este necesar ca un cuvânt să fie folosit de utilizator până când acesta v-a fi printre cele mai importante din arborele din care face parte.

Clasa Updater mai întâi interoghează dicționarul global pentru a vedea starea cuvintelor din modelul de date. Pe baza acestei interogări și din regulile conținute, setează weight-ul de actualitate și de utilizator și trimite la *dicționarul global* cuvântul împreună cu cele două weight-uri calculate.

Componența acestui modul se vede în figura 5.16

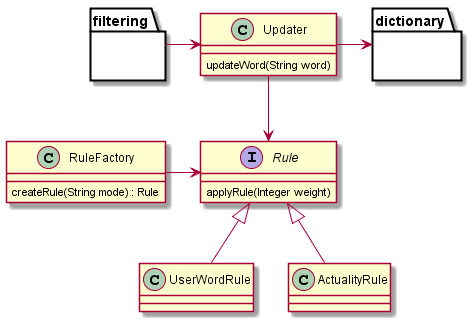


Figura 5.16 Actualizatorul de weight-uri

*Degradorul* calculează degradările weight-urilor de utilizator și de actualitate a cuvintelor folosite de utilizator. Clasa Degrador primește un semnal reprezentând faptul că o completare s-a realizat. Degrador contorizează acest lucru și interoghează fiecare cuvânt în parte, actualizând-ui aceste weight-uri.

Clasa Dictionary primește un nou parametru pentru fiecare heap care ține minte a câtea interogare se face pe acest arbore. Clasa Word primește un nou parametru care contorizează la a câta interogare s-a făcut creșterea weight-ului acestui cuvânt. Ca urmare, parametrul de timp din funcția de degradare reprezintă diferența dintre interogarea curentă pe heap-ul dicționarului și ultima interograre la care s-a făcut update la cuvânt.

Clasa Degrador are ca și componentă o funcție memorată de interfața DecayFunction. Această interfață permite schimbarea implementării funcției de degradare foarte ușor.

Figura 5.17 evidențiază toate clasele ce iau parte la crearea acestui modul.

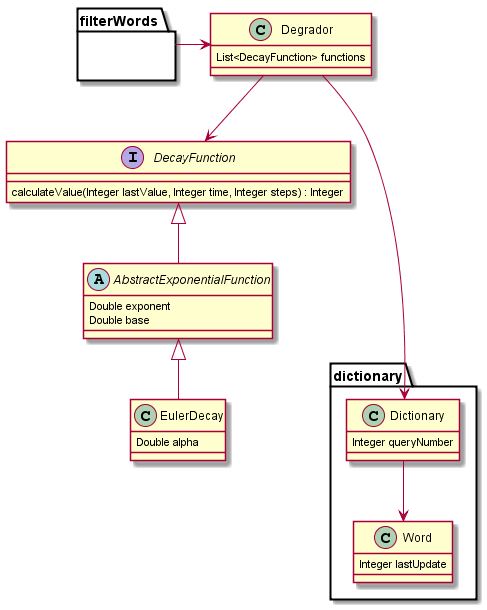


Figura 5.17. Componența degradorului

Clasa AbstractExponentialFunction este baza implementării oricărei funcții exponențiale. Formula unei astfel de funcții este dată în formula (5.1).

Clasa EulerDecay implementează această clasă și asignează fiecărui element termenul pe care în va folosi. Coeficientul este egal cu 1, baza este egală cu numărul lui euler, iar exponentul este de forma: (formula 5.2).

### Module ce se ocupă cu inițializarea și distrugerea sistemului

*Pornire* și *Terminare* sunt module alcătuite din o singură clasă. Pornire conține clasa Initializer, care se ocupă de citirea fișierului de configurări, pornește citirea dicționarului implicit și construirea modelului de date de căutare. Terminare conține clasa Terminator care apelează metoda reset() pe toate modulele din sistem. Inainte de asta, pornește salvarea dicționarului aflat curent în memorie. Astfel, sistemul este gata să se închidă.

## Vizualizarea datelor

După cum s-a explicat în această lucrare, modelul de date de căutare este o pădure de arbori Ternary Search Tree, iar modelul de date de indexare (dicționarul) este o pădure de heap-uri. Aceste modele de date sunt exemplificate în figurile 5.18 pentru datele de indexare și 5.19 pentru datele de căutare.

Să luăm ca exemplu dicționarul alcătuit din cuvintele: ”acasa=6”, ”acelasi=4”, ”alerga=10”, ”aproape=8”, ”asemenea=3”, ”astfel=1”, ”baiat=2”, ”bani=3”, ”bucati=3”, ”bine=5”, ”barca=4”, ”cutie=2”, ”cuvant=6”, ”copil=7”, ”chiar=5”, ”citit=6”. Se observă că există cuvinte ce încep cu literele ”a”, ”b” sau ”c”. Așadar, v-om avea 3 heap-uri de cuvinte și 3 arbori de căutare.

Heap-urile de cuvinte se pot vedea în figura 5.18

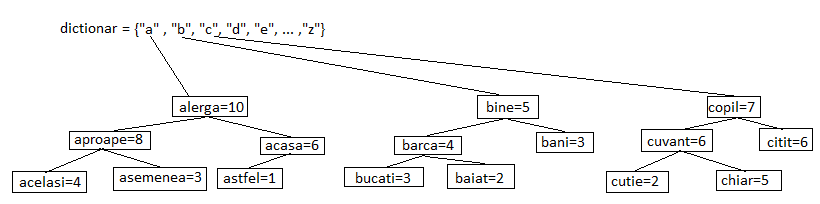


Figura 5.18. Model de dicționar

Arborii de căutare se pot vedea în figura 5.19

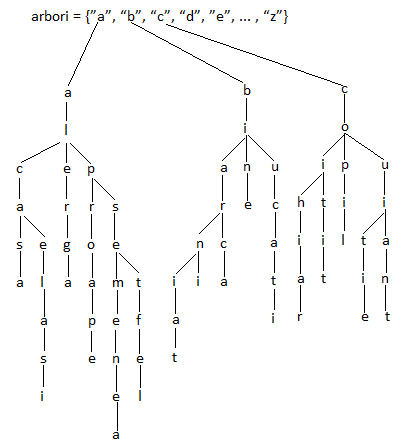


Figura 5.19. Model de arbori de căutare

# Testare şi Validare

## Scopul testelor

Soluția urmărește să se muleze cât mai ușor și rapid pe modul în care un utilizator scrie. După cum s-a explicat în secțiunea (introdu sectiunea unde se vorbeste despre constructia modelului de date cu weighturi), fiecare cuvânt își v-a actualiza parametrii de utilizator și de actualitate în momentul în care este folosit de un utilizator.

După cum reiese din figura (figura cu arhitectura conceptuala a solutiei), modul de răspuns la interogările utilizatorului este prin sugestii de completare pentru un prefix. Ca dimensiune prestabilită a acestei liste de sugestii s-a ales numărul 10. Utilizatorul nu poate urmării mai multe de atât, iar mai puține nu sunt de folos.

## Unități de măsură

O sugestie acceptată se consideră o sugestie aflată in primele 4 poziții din această listă. Acest prag delimitează sugestiile usor accesibile atât ca selectare a lor, cât și ca citire și înțelegere a lor in momentul scrierii, de cele aflate prea departe pentru a urmării cu privirea întreaga listă și selectarea lor ar dura mai mult decât scrierea în continuare a cuvântului.

Este necesară o clasificare si mai dură, deoarece o sugestie pe pozitia 1 este mult mai importantă decât una pe poziția 4, iar separarea booleană a sugestiilor nu satisface acest criteriu. Rank-ului cel mai bun al unei sugestii este acela în care se ține cont de poziție. Rankingul este similar cu ideea măsurătorilor descrise in [[18](#Placeholder1)], și anume Mean și Total Reciprocal Rank.

Metricile folosite in testare sunt :

1. Precizia – numărul de sugestii acceptate în raport cu numărul de sugestii găsite.

Având în vedere că se dorește o clasificare mai utilă decât atât și că soluția întoarce o listă cu rank-uri, se poate folosi ideea de ranking. Formula se transformă atunci în

1. Recall – numărul de sugestii acceptate în raport cu numărul de interogări

Diferența între precizie și recall este că recall-ul oferă o perspectivă asupra întregii experiențe a utilizatorului, nu doar în cazurile in care acesta găsește o sugestie potrivită.

Introducerea rank-ingului nu aduce multă valoare aceste metrici, insă consistența măsurătorilor este folositoare pentru compararea celor două mărimi.

1. Folosința utilizatorului (usefulness) – sugestii acceptate în raport cu numărul de interogări

Chiar dacă recall-ul măsoară acest lucru, această măsură păstrează formula (7.4), și reprezință cât de folositorare au fost sugestiile din punctul de vedere al utilizatorului. Utilizatorul vede doar în câte interogări a folosit sugestiile returnate.

## Descrierea modului de testare

Testarea urmărește modul in care folosința acestor indici influențează precizia modelului de date.

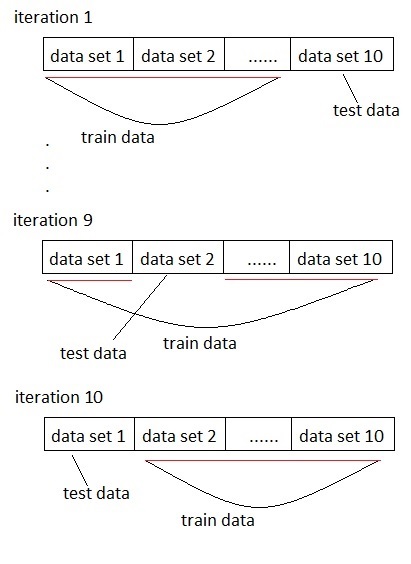
Modelul de date este supus, la fiecare test, etapei de antrenare și etapei de testare.

Etapa de antrenare presupune citirea fișierelor din care se va construi dicționarul default. Modelul va conține după această etapă toate cuvintele distincte din aceste fișiere, împreună cu weight-ul specific dicționarului default, adică frecvența de apariție a cuvântului în text.

Etapa de testare presupune interogarea modelului de date cu cuvintele prezente în fișierele de testare, simulând o interogare de utilizator. O interogare presupune interogarea modelului de date cu primele 4 litere din cuvânt ca prefix, iar apoi compararea sugestiilor returnate cu cuvântul interogat, și memorarea poziției în care s-a găsit în lista de sugestii.

Metoda de testare folosită este ten fold cross validation (figura 6.1). Toate seturile de test au fost împărțite in 10 părți și pe rând fiecare din set a fost folosit ca date de test, pe când celelate au fost folosite ca date de antrenare. La finalul testelor se face media rezultatelor. Această iterație peste fold-urile folosite ca și test este făcută pentru a nu avea un fold mai bun ca celălalt, si a obține o viziune de ansamblu a comportamentului soluției.

Pentru a putea observa că indicii de user si de actualitate se comportă conform așteptărilor, datele de test au fost rulate de mai multe ori. O rulare fără învățare presupune interogarea fără actualizarea indicilor. Prin acest lucru se observă performanța modelului de date pentru fiecare cuvânt individual. O rulare cu învățare presupune interogarea cu actualizarea indicilor, adică funcționarea normală a solutiei. Aceast tip de rulare simulează scrierea de către utilizator.

Figura 6.1. Descirerea metodei de testare Ten Fold Validation

## Descrierea seturilor de date

Toate seturile de date au fost împărțite in fișiere de câte 1000 de cuvinte, pentru fiecare set, s-au rulat testele pentru un set mic si un set mare de date de antrenare. Acest lucru arată cum influențează dimensiunea dictionarelor performanța.

Primul set de date, DocR-1,2,3, este alcătuit din mai multe documentații scrise de mine pentru facultate, în limba română. Acest set a fost testat deoarece cuvintele folosite au fost din mai multe vocabulare din mai multe domenii.

Al doilea set de date, DocE-1,2, este alcătuit din câteva documentații scrise de mine pentru facultate dar în limba engleză. Acest set a fost testat deoarece cuvintele folosite au fost dintr-un vocabular foarte restrâns.

Al treilea set de date, Mary-1,2, contine o parte din cartea *Escrocheria*, scrisă de *Mary Higgins Clark*. Acest set a fost testat pentru a urmării comportarea modelului pentru un utilizator ce scrie intr-un singur domeniu și în mod consistent folosește aceleași cuvinte din același context.

Al patrulea set de date, Paul-1,2,3, este similar cu al treilea set de date, adică o carte a unui autor. Cartea se numește *Panica din 89* scrisă de *Paul Erdman*. Pe acest set de date, pe ultima rulare, Paul-3, doar datele de test au fost luate din setul Paul. Datele de antrenare au fost luate din alți autori și din alte cărți. Scopul este de a vedea dacă contează de unde este construit dictionarul default, sau contează doar să existe cuvinte în el și acesta se va actualiza cu cuvintele utilizatorului și va continua să ruleze, cu aceleași performanțe, indiferent de dictionarul default.

Al patrulea set de date a fost folosit și pentru a demonstra variația performanțelor cu dimensiuea setului de date de input, pentru valor mici. Pentru aceasta a fost împărțit in Paul-1-1,2,3,4,5,6,7,8,9.

Al cincilea set de date, RomLit-1,2,3,4,5,6 este luat din revista românească *România Literară* și a fost folosit pentru a demonstra atât independența modelului de autorii din care provin datele de antrenare, cât și comportamentul modelului pe un set de date mare. S-au extras in mod aleator niște articole pentru a servi drep test, iar pe restul datelor, tot în mod aleator, s-au extras articole pentru a servi drept set de antrenare.

## Rezultate

Primele tabele conțin:

* Denumirea setului de date
* Dimensiunea în cuvinte a modelului de date construit din setul de date (cuvinte distincte cu weight asociat)
* Dimensiunea în cuvinte distincte a setului de testare
* Rezultatele pentru interogarea fără învățare
* Rezultatele pentru interogarea cu învățare

Primul tabel afișează informațiile pentru prima rulare a testelor. Această rulare simulează prima scriere de către utilizator. Al doilea tabel afișează a doua rulare a testelor, peste modelul de date construit din prima învățare. Această rulare simulează un al doilea document scris de către utilizator, în acest caz fiind același document.

Al treilea tabel, împreună cu reprezentarea grafică a lui de la figura 7.1, arată corelația între dimensiunea setului de antrenare și performanțele modelului de date.

Al patrulea tabel afișează dependența modelului de date pe de date de antrenare mari, cât și independența modelului de date față de autorii datelor de antrenare.

Următoarele tabele descriu rezultatele obținute pe testele efectuate:

Tabel 6.1. Prima rulare a testelor.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Set de date** | **Dim**  **antrenare** | **Dim testare** | **Interogare 1 fără învățare** | | | **Interogare 1 cu învățare** | | |
| **Precizie** | **recall** | **utilitate** | **precizie** | **recall** | **utilitate** |
| DocR-1 | 536 | 296 | **74.41** | *19.2* | 23 | **65.3** | *16.8* | 21 |
| DocR-2 | 1483 | 296 | **67.7** | *37.8* | 49 | **61.4** | *34.2* | 46 |
| DocR-3 | 2332 | 296 | **65.3** | *46.7* | 62 | **59.6** | *42.5* | 59 |
| DocE-1 | 203 | 259 | **92.2** | *26.8* | 27 | **87.2** | *25.4* | 27 |
| DocE-2 | 589 | 259 | **86.4** | *42.7* | 47 | **81.9** | *40.2* | 46 |
| Mary-1 | 651 | 369 | **85.5** | *21.8* | 24 | **80** | *20.4* | 24 |
| Mary-2 | 2057 | 369 | **80.3** | *39.1* | 46 | **76.1** | *36.9* | 45 |
| Paul-1 | 378 | 403 | **84.2** | *13* | 15 | **76.5** | *11.8* | 14 |
| Paul-2 | 2478 | 403 | **74.3** | *36.4* | 44 | **69** | *33.8* | 43 |
| Paul-3 | 7205 | 403 | **68.7** | *36.8* | 46 | **64.3** | *34.5* | 45 |

Tabel 6.2. A doua rulare a testelor

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Set de date** | **Dim antrenare** | **Dim testare** | **Interogare 2 fără învățare** | | | **Interogare 2 cu învățare** | | |
| **precizie** | **recall** | **utilitate** | **precizie** | **recall** | **utilitate** |
| DocR-1 | 536 | 296 | **77.9** | *77.5* | 94 | **70.7** | *70.3* | 91 |
| DocR-2 | 1483 | 296 | **77.9** | *77.8* | 95 | **71** | *70.9* | 92 |
| DocR-3 | 2332 | 296 | **77.8** | *77.8* | 95 | **71.2** | *71* | 91 |
| DocE-4 | 203 | 259 | **85.2** | *84.8* | 97 | **80.6** | *80.2* | 96 |
| DocE-5 | 589 | 259 | **85.3** | *85.2* | 98 | **81.2** | *81.1* | 97 |
| Mary-1 | 651 | 369 | **88.7** | *88.7* | 98 | **84.1** | *84.1* | 97 |
| Mary-2 | 2057 | 369 | **88.7** | *88.7* | 98 | **84.3** | *84.2* | 98 |
| Paul-1 | 378 | 403 | **86.1** | *86* | 98 | **81** | *80.9* | 97 |
| Paul-2 | 2478 | 403 | **86.1** | *86.1* | 98 | **81** | *81* | 97 |
| Paul-3 | 7205 | 403 | **86** | *86* | 98 | **80** | *80* | 97 |

Tabel 6.3. Variatia rezultatelor cu dimensiunea setului de date

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Set de date** | **Dim antrenare** | **Interogare 2 fără învățare** | | |
| **precizie** | **recall** | **utilitate** |
| Paul-1-1 | 378 | **84.2** | *13* | 15 |
| Paul-1-2 | 744 | **81.2** | *21* | 24 |
| Paul-1-3 | 1021 | **79.2** | *25.2* | 29 |
| Paul-1-4 | 1341 | **77** | *28.3* | 33 |
| Paul-1-5 | 1613 | **76** | *30.7* | 37 |
| Paul-1-6 | 1900 | **75.2** | *32* | 39 |
| Paul-1-7 | 2117 | **75** | *33.6* | 41 |
| Paul-1-8 | 2269 | **74.9** | *34.9* | 42 |
| Paul-1-9 | 2487 | **74.3** | *36.4* | 44 |

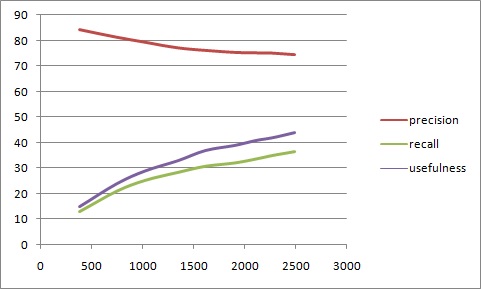


Figura 6.2. Variația metricilor cu dimensiunea setului de date

Tabel 6.4. Variația modelului de date pe date de antrenare mari cât și independența de autorii datelor de antrenare.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Set de date** | **Dim antrenare** | **Interogare 1 cu învățare** | | | **Interogare 2 cu învățare** | | |
| **precizie** | **recall** | **utilitate** | **precizie** | **recall** | **utilitate** |
| RomLit-1 | 2309 | **73.4** | *27.2* | 33 | **81** | *80.7* | 95 |
| RomLit-2 | 7459 | **67.2** | *36.5* | 46 | **80.6** | *80.3* | 95 |
| RomLit-3 | 12280 | **65.2** | *40.9* | 52 | **80.2** | *79.9* | 95 |
| RomLit-4 | 16271 | **63.8** | *42.8* | 54 | **79.9** | *79.6* | 95 |
| RomLit-5 | 25687 | **62.1** | *45.1* | 58 | **79.1** | *78.7* | 94 |
| RomLit-6 | 87122 | **60.79** | *48.7* | 63 | **78.1** | *77.6* | 94 |

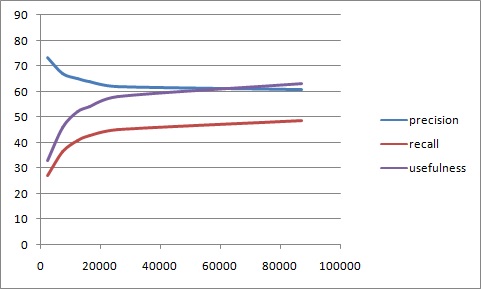


Figura 6.3. Variația metricilor cu dimensiunea setului de date (mare)

## Interpretarea rezultatelor

Modelul de date se comportă mai bine la rulările fără învățare. Acest lucru se datorează faptului că modelul este interogat pentru fiecare cuvânt în parte să se vadă cât de bine s-a actualizat modelul de date pentru utilizator. Performanțele reale sunt cele date de interogarea cu invățare. Aceste măsuri sunt mai mici, deoarece când un cuvânt este folosit de utilizator, modelul de date învață acest lucru prin actualizarea indicilor de utilizator si de actualitate. Această schimbare de indici aduce cuvântul proaspăt interogat mai sus în sugestii. Când se interoghează un alt cuvânt cu același prefix, cuvântul interogat se găsește mai jos în lista de sugestii, așadar scad metricile.

Tabelul 7.3 si reprezentarea grafică a acestuia, figura 7.2, arată corelația dintre dimensiunea dictionarului default și performanțele modelului de date la prima interogare. Precizia scade o dată cu creșterea dimensiunii dictionarului default, dar crește recall-ul și utilitatea. Acest comportament se poate interpreta în felul următor: Dacă sunt puține cuvinte, cuvântul așteptat v-a fi returnat mai sus în lista de sugestii, dar modelul nu va conține multe din cuvintele interogare te utilizator. Acest lucru se observă cel mai bine in rularile pe seturile de date Paul-1 si Paul-2.

Recall-ul crește mult de la rularea 1 la rularea 2 pentru că modelul de date a învățat cuvintele lipsă din prima rulare. Recall-ul nu este mai mare in interogarea 1 cu învățare deoarece datele de test conțin cuvinte diferite.

În ceea ce priveste dependența modelului de date de dimensiunea dictionarului, se observă că există o tendință asimptotică a valorilor metricilor de performanță. Precizia se stabilizează în jurul valorii de 60%, recall-ul tinde spre valoarea de 50% iar utilizabilitatea crește constant, în jurul valorii de 62%.

Cât despre reinterogarea datelor de test după învățarea acestora, se observă o stabilitate în jurul valorii de 79%-78% pentru precizie, o valoare de 78%-77% pentru recall și un foarte plăcut 95% pentru utilitate.

# Manual de Instalare si Utilizare

## Instalare și compilare

Ca resurse hardware pentru această aplicație este nevoie doar de un computer.

Aplicația a fost scrisă în limbajul de progamare java. Pentru a putea instala și utiliza această aplicație, instalați java versiunea 1.8.0\_45 de pe site-ul oficial oracle aici:

<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

Având în vedere că sistemul nu folosește dependențe de librării exterioare, nu este necesară instalarea de framework-uri suplimentare.

Aplicația poate fi împachetată ca un fișier jar și importată în orice aplicație.

Pentru a putea modifica codul sursă și pentru a putea recompila proiectul, este nevoie de programul IDE IntelliJ, disponibil pentru descărcare aici:

<https://www.jetbrains.com/idea/download/>

Pentru compilarea și împachetarea aplicației s-a folosit programul Maven.

<https://maven.apache.org/download.cgi>

## Utilizare

Aplicația este nu mai mult decât un jar. Pentru folosirea acesteia în cadrul unei alte aplicații, trebuie importat jar-ul în aplicație, și instanțiată interfața org.cmplhist.AutoCompletionSystem cu ajutorul Factory-ului org.cmplhist.AutoCompletionSystemFactory.

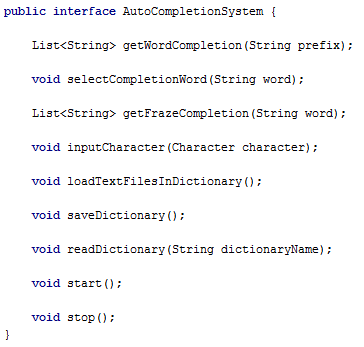


Figura 7.1 interfața sistemului de autocompletare

Pentru folosireafuncției de autocompletare de cuvinte, trebuie apelată metoda getWordCompletion cu prefixul scris de utilizator până în momentul respectiv. După ce utilizatorul scrie un cuvânt, indiferent dacă a fost selectat din lista de sugestii sau nu, clientul aplicației trebuie să apeleze metoda selectCompletionWord pentru ca sistemul să învețe cuvintele scrise de acesta.

La fiecare apăsare de caracter, clientul trebuie să apeleze metoda inputCharacter cu caracterul respectiv. Dacă nu se realizează acest lucru, autocompletarea de frază nu se v-a îmbunătății niciodată.

Pentru autocompletarea de frază clientul trebuie să apeleze metoda getFrazeCompletion cu utlimul cuvânt scris de utilizator.

Restul metodelor servesc funționalități de manipulare a dicționarelor. Salvare, încărcarea unui dicționar nou, introducerea de cuvinte în dicționar.

# Concluzii

## Recapitulare a ce reprezintă autocompletarea

Autocompletarea ca funcționalitate a plecat de la simpla sugerare de comenzi ce pot fi folosite într-un terminal linux și a ajuns să fie o funcționalitate complexă ce este abordată în unele cazuri prin structuri de date cu un fundament matematic puternic în spate sau algortmi de machine learning pentru prezicerea tot mai rapidă și mai bună a sugestiilor de completare.

Toate aceste eforturi însă se îndreaptă spre autocompletările la nivel de WSE(Web Search Engine). O aplicație stand alone pentru un calculator nu dispune de datele necesare pentru a reproduce un astfel de comportament și pentru a permite unui astfel de sistem portarea pe un calculator care nu are acces la internet.

Având în vedere acest lucru, prin această lucrare s-a urmărit analiza unui sistem care nu este orientat pe o funționare de tip ”exploratory search”. O astfel de funcționalitate este specifică WSE în care utilizatorul vrea să caute informații despre un anumit domeniu dar nu este familiar cu domeniul respectiv, sau dorește să afle, fără efort din partea lui, știri sau noutăți despre un anumit domeniu fără a cunoaște indeaproape acest domeniu.

Autocompletarea ca și funcționalitate offline este foarte folosită în domeniul programării, când programatorii au de scris multe informații repetitive, cum ar fi numele unei variabile, tipul de date al unei structuri, constante sau chiar template-uri de blocuri de tip iterativ sau condițional. Un astfel de tip de autocompletare pentru editoarele de text este un lucru foarte util pentru un utilizator.

Autocompletarea este de două tipuri: la nivel de cuvânt și la nivel de frază. Ambele tipuri de autocompletare au avantajele și dezavantajele lui. Scopul în care completarea este folositoare de asemenea este diferit de la o completare la alta. Rolul principal al autocompletării de cuvânt este acela de a reduce numărul de taste apăsate de utilizator atunci când el știe ce vrea să scrie. Rolul principal al autocompletării de frază este acela de a sugera direcții de continuare a scrierii pentru utilizator, direcții folosite de alți utilizatori sau grupări de cuvinte ce au loc des.

Problemele ridicate de autocompletare sunt atunci când dependența contextului trebuie integrată în sistem.

## Contribuții și cercetare proprie

Abordarea creări unui astfel de sistem a fost dificilă deoarece foarte multe funcționalități trebuiau create pentru ca autocompletarea să fie ceva mai mult decât căutarea unor cuvinte și returnarea acestora. Dependența de context este greu de realizat într-un sistem care nu cunoaște despre utilizator decât cuvintele pe care acesta le folosește.

O bună abordare a dependenței de context este un algoritm bazat pe ideea de inverted index. În acest tip de problemă, datele de care dispune soluția sunt cuvinte și documente, într-o pereche ordonată de tip (W,D). Dependența a mai multor cuvinte de un document reprezintă faptul că acestea fac parte dintr-un context comun. Această abordare funcționează cu succes, însă are nevoie de multe date despre utilizator (documente) pentru a putea constru o structură care să fie relevantă în multe din cazurile folosite de utilizator.

O astfel de abordare este periculoasă în cazul în care sistemul nu are cunoștiițe implicite despre un utilizator. Atunci când un utilizator dorește un sistem de autocompletare, el vrea ca acesta să funcționeze încă de la bun început și mulți utilizatori poate nu expun aceeași răbdare pentru ca sistemul să învețe din documentele pe care ei le scriu. O altă problemă este atunci când un utilizator scrie un document foarte lung, cum ar fi o carte. Cum se face împărțeala acestui document enorm în mini documente.

Datorită acestor probleme, nu am mers pe acest algoritm în construirea sistemului.

Pentru ca un sistem să răspundă rapid unui utilizator, acesta trebuie să învețe repede ce fel de cuvinte folosește utilizatorul și frecvența acestora. Pentru o astfel de abordare este necesară crearea unui model de date folosind doar cuvinte.

Prioritizarea cuvintelor este necesară. Pentru acest lucru, s-a ales asignarea de ponderi(weight) fiecărui cuvânt. Din prima observație, un weight care reflectă bine realitatea este frecvența de apariție a unui cuvânt în text. Probabilitatea ca un cuvânt cu frecvență mare de utilizare să fie folosit este mai mare decât a unui cuvânt cu frecvență mică.

Problema tratată este mai complicată. Nu toți utilizatorii scriu la fel. Unii au un vocabular mai elocvat și doresc folosirea de cuvinte academice, care nu sunt foarte frecvente în limbajul de zi cu zi. Chiar și același utilizator poate scrie într-o zi un text cu o exprimare frumoasă și în altă zi un text similar cu o exprimare mai urâtă. Un utilizator poate scrie atât un articol științific cât și poate vorbi cu familia despre vreme. Aceste diferite circumstanțe de utilizare a sistemului generează diferite contexte în care sistemul trebuie să răspundă schimbării bruște de vocabular pe care un utilizator îl folosește.

Pentru tratarea problemei folosinței mai dese a unor cuvinte puțin folosite în general s-a mai adăugat o pondere la cuvinte, și anume frecvența utilizatorului. Această pondere are un impact mai mare decât ponderea normală (implicită) a unui cuvânt, deoarece este legat de vocabularul unui utilizator. Dacă cuvintele din acest vocabular nu ar conta mai mult, atunci învățarea acestor cuvinde de către sistem poate dura mult, iar utilizatorul să nu fie satisfăcut cu acest comportament.

Lucrurile trebuie privite și invers. Dacă utilizatorul a folosit mult un anumit cuvânt acum un an de zile în perioada ce scria documente științifice, dar anul acesta s-a reprofilat pe gătit și scrie multe rețete, atunci cuvintele folosite anul trecut îi încurcă mult munca venind mai sus ca noile cuvinte despre mâncare pe care a început să le folosească recent. Este necesară așadar descreșterea frecvenței utilizatorului pentru a simula schimbarea lui de vocabular de la o perioadă la alta.

Pentru tratarea problemei schimbării dese de context în cadrul unei aceleiași zile, când utilizatorul comunică prin emailuri la servici în mod formal dar cu prietenii în mod informal este necesar ca acest lucru să se reflece în imediata folosinței unui anumit cuvânt.

Ponderea de actualitate este probabil cea mai importantă **contribuție personală** la construirea acestui sistem. Ponderea de actualitate reflectă faptul că un cuvânt este intens folosit într-un paragraf și trebuie adus în primele sugestii, dar după încetarea folosirii lui acesta trebuie să dispară.

Actualitatea este o pondere exact ca și ponderea de utilizator, adică este mai importantă ca și weight-ul implicit al unui cuvânt, și trebuie scăzută în timp pentru a reflecta schimbarea de context a utilizatorului, doar că aceste lucruri se fac mult mai brusc. Actualitatea va crește foarte rapid astfel încât de la a treia folosință a cuvântului acesta să fie în primele sugestii de completare, dar dacă trec mai multe interogări a sistemului și acest cuvânt nu mai este selectat, el trebuie să dispară. Și acest lucru se întâmplă in intervalul de câteva zeci de interogări, nu sute sau mii necesare pentru ponderea utilizator să dispară.

Un alt factor important folosit în sistem este necesitatea de rapiditate a căutărilor. Acest lucru se tratează prin folosirea unor structuri de arbori făcută special pentru stocare și interogare de șiruri de caractere.

Arborele folosit se numește ternary search tree. **Îmbunătățirile** față de expunerea din literatură se rezumă la următoarele:

* Inserarea cuvintelor în arbore după weight. Astfel, în timpul în care se face interogarea după cuvinte ce încep cu un anumit prefix se face și scoaterea lor sortate după weight, deoarece returnarea informațiilor se face în aceeași ordine în care s-au introdus.
* Adăugarea proprietății de Tournament Tree arborelui de căutare. Când se face căutarea, nodul cu greutatea subarborelui cea mai mare se parcurge primul, pentru că acolo se va găsi următorul cuvânt cu weight cel mai mare. Căutarea astfel este determinată nu exploratorie
* Împărțirea arborelui într-o pădure de arbori, astfel încât un arbore să țină minte toate cuvintele ce încep cu o anumită literă. Acest lucru s-a făcut din trei motive:
  + Scalabilitate. Sistemul nu are nevoie să aștepte după o interogare până să pornească o alta. Astfel mai multe aplicații pot cere sugestii de completare în același timp
  + Autocompletarea începe de la apăsarea primei litere, deci se știe în ce arbore trebuie căutate nodurile. Astfel nu sunt încărcate și restul datelor dacă utilizatorul vrea doar cuvinte care încep cu litera ”a”.
  + Un cuvânt este bine definit într-un arbore ca o cale de la rădăcină până la un anumit nod, nu neapărat frunză. Dacă nu era făcută această separare, mult mai multe informații despre cuvinte trebuiau ținute în arbore, informații nu foarte practice, gen de unde începe un cuvânt și care cuvânt, și unde se termină un cuvânt și care cuvânt. Astfel se memorează doar un flag că în acel nod se termină un cuvânt, și reconstrucția lui se poate face foarte ușor.

Această împărțite în arbori a fost tratată în [[2](#Kas10)], dar decizia de împărțire a fost luată înainte de citirea acetui articol. Articolul acesta doar mi-a confirmat faptul că această decizie a fost una inspirată

Ca testare a sistemului, singura metodă este de a observa acești indici de actualitate și de utilizator în acțiune. Un astfel de test care să demonstreze acest lucru trebuie să interogheze un sistem cu aceleași date de mai multe ori pentru a observa creșterea preciziei sistemului.

## Testare

Rezultatele testelor sunt promițătoare. Se observă în tabelul 6.1 la o primă rulare a setului de test, că rezultatele preciziei se învârt în cel mult jurul valorii de 80% pentru seturi de date favorabile, pot scădea până la 60 la sută pentru seturi de date nefavorabile , însă media este undeva la 70%. Despre recall datele sunt și mai proaste, obținând un maxim de 42% și un minim de 11%, cu o medie de 28%. Utilitatea sistemului este apropiată de recall, cu câteva procente în plus, situându-se la 59% maxim, 14% minim și 37% media.

Un astfel de comportament lasă de dorit din ceea ce privește perspectiva utilizatorului. Pentru un utilizator să obțină un recall de 11 la sută înseamnă că aproape nici un cuvânt interogat de el nu era memorat în dicționar, dar și din cele existente precizia să fie atât de mică înseamnă că utilizatorul ar putea folosi sistemul la 1 din 15 cuvinte. Viteza de scriere nu crește aproape deloc.

Pentru al doilea set de teste, când sistemul a introdus aceste ponderi pentru cuvintele interogate, creșterea este consistentă peste toate rezultatele, așa cum se arată în tabelul 6.2. Toate cuvintele au fost introduse în dicționar, recall-ul este egal cu precizia, ceea ce zice că nu mai avem interogări ce să nu returneze un rezultat, iar rezultatele returnate sunt dintre cele mai bune.

Avem ca precizie o valoare de 84% maximă, 70% minimă cu o medie de 81%. Recallul este la fel, iar utilitatea sistemului se află în intervalul 98% - 91% cu o valoare medie de 96%. Asta înseamnă că în 19 din 20 de interogări, utilizatorul a avut cuvântul căutat în primele 4 sugestii.

Sistemul se comportă decent și când vine vorba de volum mare de cuvinte aflate în dicționar. Pentru un set de date de 80.000 de cuvinte în dicționar, la o primă interogare se obțin valori de 60% pentru precizie, 48% pentru recall și 62% utilitate. Recall-ul este scăzut deoarece, fiind atât de multe cuvinte, cuvântul căutat nu se află printre sugestiile returnate de interogare.

La o a doua interogare se observă aceeași creștere a tuturor metricilor, precizia ajungând la 78%, recall-ul la 77.6% și utilitatea la 94%.

## Dezvoltări ulterioare

Sistemului îi lipsește integrarea cuvintelor în context. O bună încercare de a intergra un simulator de context este următoarea: la fiecare cuvânt folosit, înainte de inceperea interogării pentru cucînt nou sistemul să crească weight-urile cuvintelor ce urmează după cuvântul folosit în structura de completare de frază. Astfel cuvintele corelate o să primească tot timpul un weight mai mare.

# Bibliografie

x

|  |  |
| --- | --- |
| 1 | Ward, David, Hahn, Jim, and Feist, Kirsten. Autocomplete as a Research Tool: A Study on Providing Search Suggestions. *INFORMATION TECHNOLOGY AND LIBRARIES* (DECEMBER 2012). |
| 2 | Kastrinakis, Dimitrios and Tzitzikas, Yannis. Advancing Search Query Autocompletion Services with More and Better Suggestions. *Lecture Notes in Computer Science*, 6189 (2010), 35-49. |
| 3 | Nandi, Arnab and Jagadish, H. V. Assisted Querying Using Instant-Response Interfaces. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of data* (New York 2007). |
| 4 | Jung, Hanmin et al. Comparative Evaluation of Reliabilities on Semantic Search Functions: Auto-complete and Entity-centric Unified Search. In *Proceedings of the 5th International Conference on Active Media Technology* (Berlin 2009). |
| 5 | Wu, Hao. Search-As-You-Type in Forms: Leveraging the Usability and the Functionality of Search Paradigm in Relational Databases. *36th International Conference on Very Large Data Bases* (september 13-17, 2010), 36-41. |
| 6 | Paek, Tim, Lee, Bongshin, and Thiesson, Bo. Designing Phrase Builder: A Mobile Real-Time Query Expansion Interface. In *Proceedings of the 11th International Conference on Human-Computer Interaction with Mobile Devices and Services* (New York 2009). |
| 7 | Jacquet, Philippe and Regnier, Mireille. Trie partitioning process: Limiting distributions. In *Springer Berlin* (Heidelberg, 1986). |
| 8 | Baberwal, Sanjay and Choi, Ben. Speeding up keyword search for search engines. *3rd IASTED International Conference on Communications, Internet, and In-* (November 2004), 255-260. |
| 9 | Matani, Dhruv. *An O(k log n) algorithm for prefix based ranked autocomplete*. 2011. |
| 10 | Chaudhuri, Surajit and Kaushik, Raghav. Extending Autocompletion To Tolerate Errors. In *SIGMODț09* (Providence, Rhode Island June 29–July 2, 2009). |
| 11 | Jurafsky, Dan and Martin, James H. *Speech and Language Processing: An Introduction to Natural Language Processing, Speech Recognition, and Computational Linguistics. 2nd edition*. Prentice-Hall, Standford, 2009. |
| 12 | Bast, Holger and Weber, Ingmar. Type Less, Find More: Fast Autocompletion Search with a Succinct Index. *SIGIR'06* (August 6-11, 2006). |
| 13 | Witten, I.H., Bell, T.C, and Moffat, A. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, 1999. |
| 14 | Bast, Holger, Mortensen, Christian W., and Ingmar, Weber. Output-Sensitive Autocompletion Search. |
| 15 | Bar-Yossef, Ziv and Kraus, Naama. Context-Sensitive Query Auto-Completion. *WWW 2011* (March 28-Aplri 1 2011). |
| 16 | Nandi, Arnab and Jagadish, H.V. Effective phrase prediction. In *VLDB* ( 2007). |
| 17 | Knuth, D. E. *The Art of Computer Programming, Volume 3: Sorting and Searching. 2nd edition*. Addison-Wesley Publishing Company, 1998. |
| 18 | Hinze, Ralf. *A Simple Implementation Technique for Priority Search Queues*. Institute of Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands, 2001. |
| 19 | Freeman, Eric, Freeman, Elisabeth, Sierra, Kathy, and Bates, Bert. *Head First Design Patterns, First Edition*. O'Reilly Media, Inc, 2004. |
| 20 | D. Radev, H. Qi, H. Wu and W. Fan. Evaluating Web-based Question Answering Systems. In *Proceedings of LREC* ( 2002). |
| 21 | Bang-Jensen, Jørgen and Gutin, Gregory. *Digraphs: Theory, Algorithms and Applications*. Springer, 2000. |
| 22 | Leike, A. Demonstration of the exponential decay law using beer froth. *European Journal of Physics*, 23, 1 (2002), 21-30. |
| 23 | Radev, D., H., H.n, W. Evaluating Web-based Question Answering Systems. In *Proceedings of LREC* ( 2002). |
| 24 | Goldstein, Lay, Schneider, and Asmar. *Brief calculus and its applications, 11th ed.* Prentice–Hall, 2006. |
| 25 | Bentley, J and Sedgewick, B. Fast Algorithms for Sorting and Searching Strings (april 1, 1998). |
|  |  |

x