Arhivare Digitală și Căutare Eficientă

Andreea Nițu

April 2024

Sinopsis

Avansul tehnologiei în viețile noastre are nenumărate beneficii, de la educație, până la sănătate și transport. Acest avans a luat amploare aproape în toate domeniile odată cu apariția mașinilor de calcul ce au automatizat majoritatea proceselor. Totodată, creșterea aceasta bruscă a mediului digital și a masei informaționale a determinat o nevoie vitală de noi metode, mai rapide și mai sigure, de a accesa toată această informație. Prima pagină web de pe Internet avea o dimensiune de doar 4 KB pe când o pagină web obișnuită avea în 2014 în jur de 1.2 MB, o creștere exponențială de 29 900% [1]. Din 2014 până în prezent s-a înregistrat o creștere de 134% a dimensiunii unei pagini web pentru Desktop și 300% pentru mobil [2].

Tendința pieței și a modului de dezvoltare de software actual este acela de a implementa un algoritm ce întoarce un rezultat în cel mai scurt timp de la rulare, compromisul fiind acela că va necesita mai multă memorie, atât volatilă cât și de stocare.

Din aceste considerente, problemele pe care le abordez sunt legate de manipularea eficientă de informații, în cazul de față de tip text, și compararea metodei găsite cu soluții naive sau alte soluții folosite. Această problemă se mulează perfect pe cazul unei biblioteci arhivate digital, ce conține diverse documente, cărți, etc., ce se doresc a fi folosite ulterior pentru a căuta informații. Pentru a putea genera textul conținut în aceste documente, am testat multiple soluții de OCR, iar pentru căutarea în aceste fișiere rezultate am implementat un algoritm eficient de indexare și căutare a textului.

Cuprins

Capitolul 1

Introducere

1.1 Povestea din spate

Creșterea cantității de informație distribuită în rețea a determinat atât dezvoltarea infrastructurii fizice, conexiune mai rapidă la Internet datorită fibrei optice ajungându-se la o viteză standard de un gigabit pe secundă transfer de date, cât și dezvoltarea software-ului necesar manipulării tuturor acestor date. Fie refolosiți și îmbunătățiți, fie descoperiți noi algoritmi și tipuri de date ce permit stocarea, modificarea și căutarea datelor într-un mod cât mai eficient și rapid posibil.

Situația actuală a cantității uriașe de informații cu care se confruntă orice companie ce produce software împinge dezvoltarea către un loc în care fiecare milisecundă contează. Limbajele de nivel înalt (precum Python sau Ruby) nu se mai folosesc pentru orice scop. De exemplu, Python este mai des întâlnit acum în cadrul aplicațiilor ce folosesc modele de machine learning datorită ușurinței sale în utilizare. În cadrul acestor tipuri de software, logica și corectitudinea algoritmilor sunt mult mai importante decât viteza.

1.2 Context

Conceptul de Space-Time Tradeoff se referă la faptul că, în contextul relației timp-memorie a unui program, una dintre cele două este "sacrificată" pentru a o putea obține pe cealaltă. Timpul și spațiul necesar execuției pot fi asemănate unei balanțe, fiecare dintre cele două aflându-se pe unul dintre talere. Atunci când timpul de rulare crește, eficiența de spațiu va fi mai bună, mai exact, atunci când rezultatul se recalculează de fiecare dată, memoria este mai puțin accesată. Dacă timpul de execuție este mic, memoria va fi mai intens utilizată, datorită structurilor de date sau a modului de organizare a informațiilor care salvează rezultate obținute anterior. În mod ideal, ambele ar trebui să ajungă la nivel minim, însă acest lucru nu este posibil.

Pentru această lucrare, am ales un caz concret și des întâlnit în care am aplicat conceptul de Space-Time Tradeoff și am dezvoltat un algoritm ce întoarce în timp liniar rezultatul unei căutări de text într-o colecție de documente. Acest caz simulează o bibliotecă sau o instituție care dorește să arhiveze documente din format fizic în format digital și să poată utiliza o funcție de căutare de text în toate aceste documente.

Obiectivul temei alese este acela de a găsi o metodă eficientă de căutare, ce va fi folosită fie de clienții bibliotecii, fie de personalul instituției, care să returneze rezultate într-un timp foarte scurt dintr-o colecție foarte mare de documente. Am ilustrat cum, folosind mai multă memorie, am ajuns la un timp de căutare efectivă foarte bun, ce se potrivește ideal într-un astfel de context. Acest lucru implică folosirea unei structuri de date ce reține informații despre fiecare cuvânt în parte, dar și despre cuvinte care îl precedă pe acesta. Folosind această metodă, am comparat timpii de căutare relativ la alte metode și am ajuns la rezultate exponențial mai bune.

1.3 Structura lucrării

În această lucrare voi prezenta atât conceptele teoretice din spatele implementării mele, cât

și rezultatele obținute din testele efectuate. Voi începe prin a prezenta structura aplicației și a discuta despre tehnologiile utilizate și motivația din spatele alegerilor făcute. Voi include explicații privind împărțirea aplicației în servicii, fiecare jucând un rol separat dar și legăturile dintre aceste servicii.

Apoi, voi discuta despre serviciul ce se ocupă de organizarea fișierelor și extragerea textului din imagini folosind OCR. În cadrul acestui segment, voi prezenta diferitele biblioteci de OCR pe care le-am folosit, rezultatele obținute în urma testelor efectuate asupra lor, cât și o scurtă prezentare teoretică a fiecăreia.

Serviciul ce se ocupă cu încărcarea documentelor în aplicație, dar și organizarea acestora în fișiere va fi descris pe scurt, acesta jucând un rol de interfață cu utilizatorul mai mult decât să facă parte din logică în sine. În cadrul aceleiași aplicații, se află și locul unde utilizatorul va căuta textul dorit și va primi rezultatele.

Ultimul serviciu este acela ce se ocupă de indexarea efectivă a documentelor. În acest capitol, voi discuta despre noțiunile teoretice din spatele structurilor de date alese, voi prezenta algoritmii de indexare și căutare implementați și voi demonstra faptul că acești algoritmi au o complexitate foarte bună. În finalul acestui capitol se va afla secțiunea de teste legate de eficiența superioară a algoritmilor aleși în comparație cu alte soluții.

Capitolul de concluzii va cuprinde trăsăturile bune ale soluției propuse, reieșite din teste și posibilități de dezvoltare ulterioară. Voi compara mai în detaliu soluțiile existente cu proiectul dezvoltat de mine.

1.4 Motivația lucrării

Motivația personală a soluției propuse este aceea că îmi doresc un mod eficient și ușor de utilizat prin care îmi pot arhiva digital și căuta informațiile din cărți, reviste și alte lucrări tipărite pe care le dețin. Îmi doream în principal să folosesc o astfel de aplicație în scopul documentării cercetărilor științifice la care voi lucra în viitor.

În context științific, această aplicație abordează problema căutării eficiente a unei fraze într-un text. În cazul de față, textul este reprezentat de multiple documente, fiecare document fiind organizat în pagini care apoi sunt indexate. În cadrul acestei lucrări, se dorește în principal optimizarea căutării, ajungând la valori ale complexității din punct de vedere al timpului de O(Max(n, L)), unde n este numărul de cuvinte al frazei căutate și L este adâncimea maximă a arborelui asociat fiecărei intrare din index.

Capitolul 2

Metode Existente

2.1 Prezentarea metodelor existente

Conceptul de a căuta un element specific într-o colecție mai mare de obiecte este cunoscut sub numele de "needle in the haystack" (acul în carul cu fân). În cazul de față se dorește căutarea unei secvențe de cuvinte într-un text mai mare. Există multe metode de a găsi răspunsul la această problemă, fiecare având puncte forte și slabe, putând fi utilizate în cazuri specifice. În cele ce urmează voi trece printr-o serie de algoritmi cu care am comparat funcția de căutare a algoritmului meu și îi voi descrie pe aceștia atât ca mod de funcționare, cât și complexitate. Voi presupune de la început că n este lungimea textului în care se realizează căutarea iar m este lungimea șablonului pentru care se realizează această căutare. [3]

Cea mai naivă metodă de a căuta o frază într-un text este de a lua textul cuvânt cu cuvânt și, în momentul în care primul cuvânt din frază se potrivește cu cuvântul curent, se verifică și următoarele cuvinte din frază. Desigur, această metodă este foarte ineficientă și, pentru cantități mari de date, este și impractică. Complexitatea acestui algoritm este (n+m) în medie și (n*m) în cel mai rău caz.

Una dintre cele mai cunoscute metode de căutare de pattern-uri într-un string se numește Regex (regular expression). Această metodă a fost introdusă de către Stephen Cole Kleene și datează încă din anul 1951, sub forma sa matematică [4]. Conceptul de Regex este folosit astăzi sub foarte multe forme, cele mai multe în cadrul algoritmilor de căutare de text sau chiar în analiza semantică a limbajelor de programare în cadrul compilării.

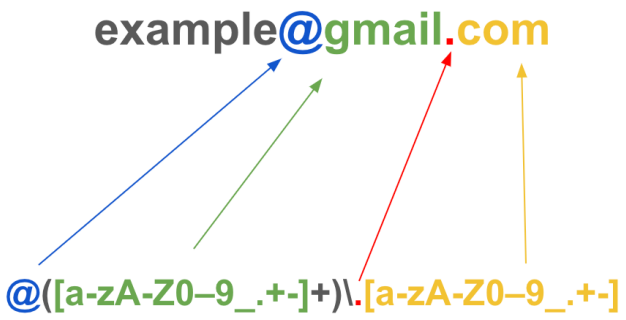


Fig. 1 - Exemplu de expresie regulată (Regex) [5]

Paradigma funcționează pe baza unor reguli pe care textul ce se dorește a fi căutat trebuie să le respecte. Fiecare caracter dintr-o expresie regulată trebuie să fie un metacaracter ce are o însemnătate anume sau un caracter normal ce se dorește a fi căutat. Spre exemplu, în cadrul expresiei "b.", caracterul "b" este un caracter simplu ce se dorește a fi căutat, în timp ce "." este un metacaracter ce semnifică faptul că după un caracter "b" poate urma orice alt caracter, în afară de new line. Pattern-urile obținute folosind regulile Regex sunt utilizate în cadrul unui procesator de Regex, care traduce expresiile regulate în reprezentări interne care mai apoi pot fi utilizate în diverse aplicații. Una dintre aceste reprezentări interne poate fi, conform algoritmului de construcție Thompson, un automat finit nedeterminist care, mai apoi, este convertit într-un automat finit determinist ce poate fi folosit pentru căutare de text.

Un automat finit este un model matematic și un concept cheie în informatică și teoria limbajelor formale. Reprezintă un dispozitiv abstract cu o stare inițială, un set de stări posibile, un set de tranzacții între stări pe baza intrărilor și o stare finală sau mai multe stări finale în care procesul se oprește sau produce o ieșire specifică. Un automat finit determinist (AFD)

este un tip de automat finit în care pentru fiecare stare și fiecare simbol din alfabet există o singură tranziție posibilă. Cu alte cuvinte, în orice moment și pentru orice simbol de intrare, AFD-ul poate lua o singură decizie cu privire la starea următoare.

În schimb, un automat finit nedeterminist (AFN) poate avea mai multe tranzacții posibile pentru aceeași stare și același simbol de intrare. Acest lucru îi conferă o mai mare flexibilitate în expresivitate, deoarece poate să exploreze mai multe ramuri ale posibilelor căi în paralel. Cu toate acestea, acest lucru poate duce și la o complexitate mai mare în implementare și în evaluarea automatului. Diferența principală între cele două tipuri de automate este în modul în care decid tranzițiile între stări. AFD-ul are o tranziție unică și clară pentru fiecare simbol de intrare, în timp ce AFN-ul poate alege între mai multe tranzacții posibile, putându-se bloca temporar sau face alegeri nedeterministe. Pentru a putea utiliza regulile definite în expresiile regulate, acestea trebuie să fie transformate în automat finit determinist, deoarece majoritatea algoritmilor de căutare și procesare a textului funcționează mai eficient cu AFD-uri.

Un automat finit determinist (AFD) poate fi folosit pentru a implementa un motor de căutare bazat pe Regex. Acesta poate să parcurgă textul caracter cu caracter, având o singură cale posibilă în funcție de simbolul curent și starea curentă. De exemplu, în expresia regulată "abc", un AFD va trece prin stări distincte pentru literele "a", "b" și "c", respectând o ordine strictă și fără să facă alegeri multiple.

Pe de altă parte, expresiile regulate pot fi traduse și în automate finite nedeterministe (AFN), care sunt mai expresive în ceea ce privește posibilitățile de potrivire a textului.

În expresiile regulate, metacaracterul "" indică faptul că simbolul anterior poate apărea de zero sau mai multe ori. De exemplu, în expresia "ab", un AFN ar putea să exploreze mai multe căi posibile pentru potrivirea textului, de la zero "a" la oricâte "a" consecutive urmate de un "b".

Deși AFN-urile sunt mai flexibile și pot descrie expresii regulate mai complexe, implementarea lor este mai dificilă și necesită algoritmi speciali pentru evaluarea și potrivirea textului. Din acest motiv, în practică, expresiile regulate sunt de obicei transformate în automate finite deterministe (AFD) pentru o eficiență mai mare în procesarea textului în cadrul procesatorului de Regex. Această conversie se face folosind algoritmi precum algoritmul lui Thompson, care transformă expresiile regulate în AFD-uri echivalente și optimizate pentru căutare și potrivire. Complexitatea pattern matching-ului realizat de către procesatoare este destul de greu de calculat pentru un Regex oarecare. Conform [4], Regex are o complexitate din punct de vedere al timpului de O(n), însă construcția automatului finit determinat are un cost de $O(2^{\hat{m}})$. Această valoare a complexității provine din necesitatea repetată de a transforma reprezentarea sub formă de AFN într-un AFD, lucru ce necesită o cantitate de timp exponențială în funcție de numărul de stări și tranzacții ale AFN-ului inițial. Mai exact, complexitatea timpului pentru transformarea unui AFN într-un AFD poate fi în O(2n), unde n reprezintă numărul de stări din AFN. Motivul principal al acestei valori provine din faptul că, în general, AFN-urile pot avea multiple tranzacții posibile pentru aceeași intrare și aceeași stare. Transformarea într-un AFD necesită explorarea tuturor combinațiilor posibile de stări, inclusiv cele care sunt rezultatul unei alegeri nedeterministe din AFN. Acest lucru poate duce la o creștere exponențială a numărului de stări în AFD în comparație cu AFN-ul inițial.

Paradigma funcționează pe baza unor reguli pe care textul ce se dorește a fi căutat trebuie să le respecte. Fiecare caracter dintr-o expresie regulată trebuie să fie un metacaracter ce are o însemnătate anume sau un caracter normal ce se dorește a fi căutat. Spre exemplu, în cadrul expresiei "b.", caracterul "b" este un caracter simplu ce se dorește a fi căutat, în timp ce "." este un metacaracter ce semnifică faptul că după un caracter "b" poate urma orice alt caracter, în afară de new line. Pattern-urile obținute folosind regulile Regex sunt utilizate în

cadrul unui procesator de Regex, care traduce expresiile regulate în reprezentări interne care mai apoi pot fi utilizate în diverse aplicații. Una dintre aceste reprezentări interne poate fi, conform algoritmului de construcție Thompson, un automat finit nedeterminist care, mai apoi, este convertit într-un automat finit determinist ce poate fi folosit pentru căutare de text.

Un automat finit este un model matematic și un concept cheie în informatică și teoria limbajelor formale. Reprezintă un dispozitiv abstract cu o stare inițială, un set de stări posibile, un set de tranzacții între stări pe baza intrărilor și o stare finală sau mai multe stări finale în care procesul se oprește sau produce o ieșire specifică. Un automat finit determinist (AFD) este un tip de automat finit în care pentru fiecare stare și fiecare simbol din alfabet există o singură tranziție posibilă. Cu alte cuvinte, în orice moment și pentru orice simbol de intrare, AFD-ul poate lua o singură decizie cu privire la starea următoare.

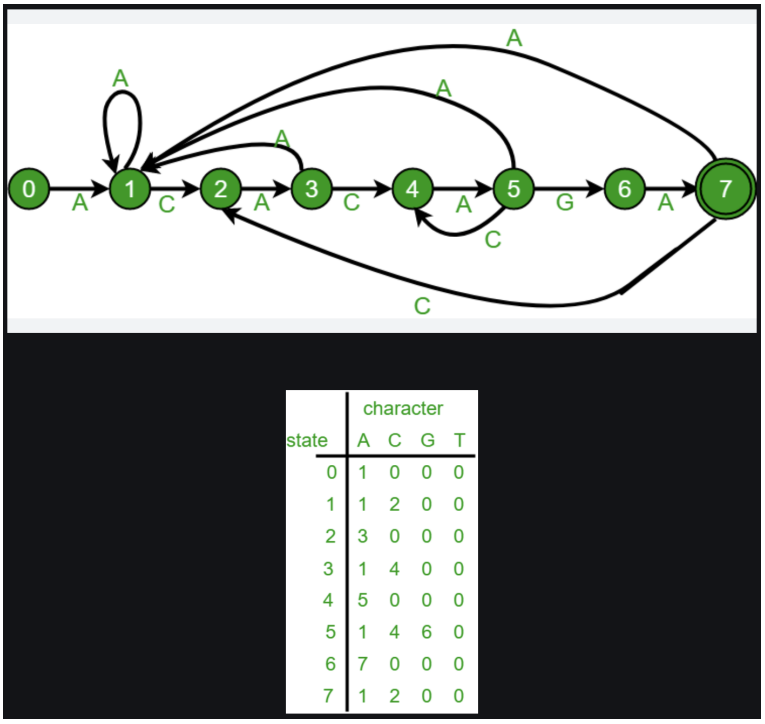
În schimb, un automat finit nedeterminist (AFN) poate avea mai multe tranzacții posibile pentru aceeași stare și același simbol de intrare. Acest lucru îi conferă o mai mare flexibilitate în expresivitate, deoarece poate să exploreze mai multe ramuri ale posibilelor căi în paralel. Cu toate acestea, acest lucru poate duce și la o complexitate mai mare în implementare și în evaluarea automatului. Diferența principală între cele două tipuri de automate este în modul în care decid tranzițiile între stări. AFD-ul are o tranziție unică și clară pentru fiecare simbol de intrare, în timp ce AFN-ul poate alege între mai multe tranzacții posibile, putându-se bloca temporar sau face alegeri nedeterministe. Pentru a putea utiliza regulile definite în expresiile regulate, acestea trebuie să fie transformate în automat finit determinist, deoarece majoritatea algoritmilor de căutare și procesare a textului funcționează mai eficient cu AFD-uri.

Un automat finit determinist (AFD) poate fi folosit pentru a implementa un motor de căutare bazat pe Regex. Acesta poate să parcurgă textul caracter cu caracter, având o singură cale posibilă în funcție de simbolul curent și starea curentă. De exemplu, în expresia regulată "abc", un AFD va trece prin stări distincte pentru literele "a", "b" și "c", respectând o ordine strictă și fără să facă alegeri multiple. Pe de altă parte, expresiile regulate pot fi traduse și în automate finite nedeterministe (AFN), care sunt mai expresive în ceea ce privește posibilitățile de potrivire a textului.

În expresiile regulate, metacaracterul "" indică faptul că simbolul anterior poate apărea de zero sau mai multe ori. De exemplu, în expresia "ab", un AFN ar putea să exploreze mai multe căi posibile pentru potrivirea textului, de la zero "a" la oricâte "a" consecutive urmate de un "b".

Deși AFN-urile sunt mai flexibile și pot descrie expresii regulate mai complexe, implementarea lor este mai dificilă și necesită algoritmi speciali pentru evaluarea și potrivirea textului. Din acest motiv, în practică, expresiile regulate sunt de obicei transformate în automate finite deterministe (AFD) pentru o eficiență mai mare în procesarea textului în cadrul procesatorului de Regex. Această conversie se face folosind algoritmi precum algoritmul lui Thompson, care transformă expresiile regulate în AFD-uri echivalente și optimizate pentru căutare și potrivire. Complexitatea pattern matching-ului realizat de către procesatoare este destul de greu de calculat pentru un Regex oarecare. Conform [4], Regex are o complexitate din punct de vedere al timpului de O(n), însă construcția automatului finit determinat are un cost de O(2m). Această valoare a complexității provine din necesitatea repetată de a transforma reprezentarea sub formă de AFN într-un AFD, lucru ce necesită o cantitate de timp exponențială în funcție de numărul de stări și tranzacții ale AFN-ului inițial. Mai exact, complexitatea timpului pentru transformarea unui AFN într-un AFD poate fi în O(2n), unde n reprezintă numărul de stări din AFN. Motivul principal al acestei valori provine din faptul că, în general, AFN-urile pot avea multiple tranzacții posibile pentru aceeași intrare și aceeași stare. Transformarea într-un AFD necesită explorarea tuturor combinațiilor posibile de stări, inclusiv cele care sunt rezul-

tatul unei alegeri nedeterministe din AFN. Acest lucru poate duce la o creștere exponențială a numărului de stări în AFD în comparație cu AFN-ul inițial.



state	character A	C	G	T
0	1	0	0	0
1	1	2	0	0
2	3	0	0	0
3	1	4	0	0
4	5	0	0	0
5	1	4	6	0
6	7	0	0	0
7	1	2	0	0

Fig. 2 - Exemplu de automat finit pentru un pattern [6]

Unul dintre cei mai eficienți algoritmi de căutare a unui substring este Boyer-Moore, dezvoltat de către Robert Boyer și Strother Moore în 1977 [7]. Comparativ cu algoritmul naiv de căutare unde se testează caracter cu caracter dacă textul căutat și textul în care se caută sunt la fel și, în cazul în care unul din caractere diferă, se trece la următorul caracter, Boyer-Moore sare peste mai mult de un singur caracter. O altă diferență este aceea că acest algoritm compară caractere începând de la dreapta la stânga, adică începând cu ultimul caracter din șirul căutat. Algoritmul dispune de două euristici ce tratează diferit modul în care se sare peste caractere, iar cel mai bun rezultat se obține combinând aceste două euristici și sărind peste numărul maxim de caractere obținut din utilizarea lor. Pentru simplitatea explicației, vom numi P șirul căutat, T textul în care se caută și c caracterul curent din T. Euristica caracterului incorect ("bad character heuristic") se aplică în momentul în care c este diferit de un caracter din P, sărind peste atâtea spații astfel încât c să fie aliniat cu ultima apariție a sa în P. În cazul în care c nu se află în P, vom sări atâtea caractere cât lungimea șirului P. În cel mai bun caz, algoritmul are o complexitate O(n/m) în cazul în care șirul P nu se află în T.

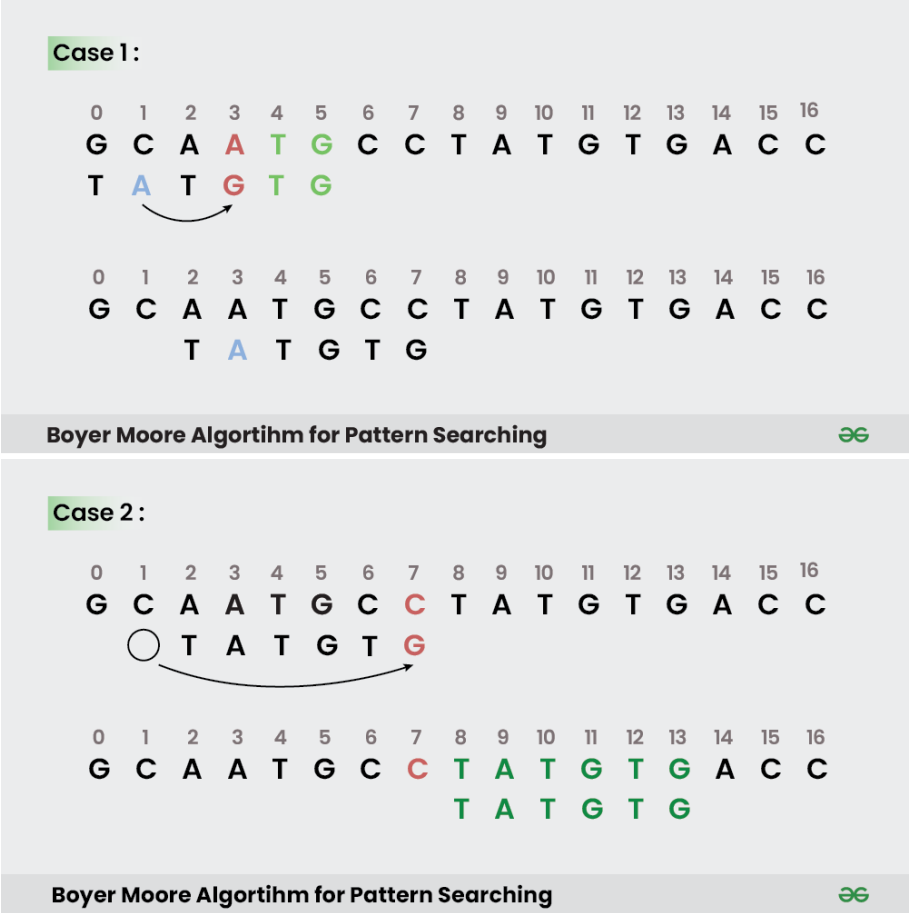


Fig. 3 - Exemple de Boyer-Moore pentru euristica caracterului incorect [8]

A doua euristică se numește euristica sufixului bun ("good suffix heuristic") și se concentrează pe găsirea unei secvențe de caractere de lungime t, numită sufix bun, care se potrivește cu un sufix al șablonului căutat. Această euristică are nevoie de un pas de preprocesare deoarece este nevoie de două tabele, unul pentru a fi folosit într-un caz general, iar celălalt în cazul în care nu se returnează niciun rezultat relevant [9].

i	0	1	2	3	4	5	6	7	8	9	10
T	A	B	A	A	B	A	B	A	C	B	A
P	C	A	B	A	B						

i	0	1	2	3	4	5	6	7	8	9	10
T	A	B	A	A	B	A	B	A	C	B	A
P			C	A	B	A	B				

Figure – Case 1

Fig. 4 - Exemplu de Boyer-Moore pentru euristica sufixului bun [9]

Din punct de vedere al complexității, algoritmul Boyer-Moore execută căutarea efectivă într-un timp de O(n*m), în cel mai rău caz și (n/m) în cel mai bun caz, adăugându-se partea

de preprocesare necesară celei de-a doua euristici de O(m+k), unde k este dimensiunea alfabetului.

Algoritmul Z este un algoritm de căutare a șablonului în șiruri de caractere care operează în timp liniar, complexitatea timp fiind O(m+n) [10]. Funcționarea sa se bazează pe conceptul de construire a unei tabele Z pentru șablonul dat și textul în care căutăm. Tabela Z stochează pentru fiecare poziție i din șir o valoare Z[i] care reprezintă lungimea celui mai lung prefix comun între șablon și sufixul șirului începând de la poziția i [11].

Algoritmul Z începe prin a combina șablonul și textul într-un singur șir, delimitat de un caracter special care nu apare în niciuna dintre cele două. Apoi, este construită tabela Z pentru acest șir combinat. Prin parcurgerea acestei tabele, algoritmul identifică toate pozițiile în care valoarea Z corespunzătoare unei poziții din șablon este egală cu lungimea șablonului însuși. Aceste poziții reprezintă locațiile în care șablonul se potrivește cu textul și sunt identificate în timp liniar, fără a repeta comparațiile inutile. Acest algoritm este asemănător algoritmului KMP despre care voi vorbi în continuare și din acest considerent nu a fost ales pentru compararea performanțelor cu algoritmul dezvoltat de mine.

Algoritmul Knuth-Morris-Pratt (KMP) este un alt algoritm eficient pentru căutarea șabloanelor în șiruri de caractere, având o complexitate de timp liniară în raport cu lungimea textului în care se face căutarea și lungimea șablonului. Performanța algoritmului în cel mai rău caz se rezumă la (m) pentru preprocesare și (n) pentru căutare în sine [12]. Ideea de bază a algoritmului este construirea unei tabele de prefixe pentru șablon, care este utilizată pentru a evita comparațiile inutile atunci când se efectuează căutarea în text.

În prelucrarea inițială a șablonului, se creează o tabelă de prefixe care indică cea mai mare lungime a unui sufix propriu care este și prefix al șablonului. Această informație este apoi folosită pentru a determina cum să deplaseze șablonul în timpul căutării. Atunci când o potrivire eșuează la o anumită poziție i din text, algoritmul KMP utilizează informațiile din tabela de prefixe pentru a calcula o deplasare optimă a șablonului astfel încât să evite comparațiile care nu vor returna rezultate relevante [13].

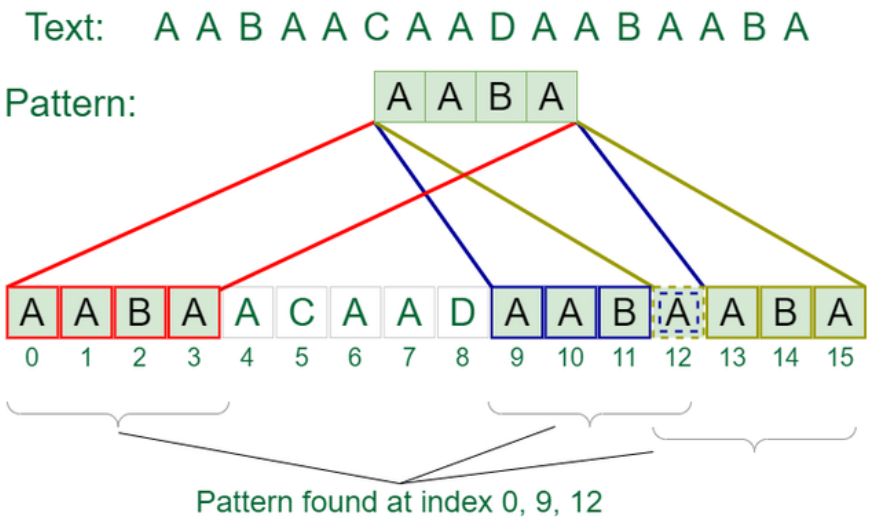


Fig. 5 - Exemplu de KMP pe un text în care cuvântul căutat apare de mai multe ori

Algoritmul Rabin-Karp este o altă tehnică eficientă de căutare a șabloanelor în șiruri de caractere, folosind tehnici de hashing pentru a identifica rapid potențiale potriviri între șablon

și subșiruri din text. Acest algoritm calculează valorile hash pentru șablon și pentru subșirurile consecutive din text, comparând aceste valori pentru a determina dacă există o potrivire sau nu.

În etapa de preprocesare, algoritmul Rabin-Karp calculează valoarea hash a șablonului și a primelor m caractere din text, unde m este lungimea șablonului. Apoi, algoritmul compară valorile hash. Dacă valorile hash coincid, se efectuează o verificare detaliată a potrivirii caractere cu caracterul din text și șablon. Dacă valorile hash nu coincid, algoritmul se deplasează la următoarea subșir din text și recalculează valoarea hash pentru acea secvență.

Complexitatea algoritmului este, în cel mai rău caz, O(m*n) în cazul căutării și O(m) în cazul preprocesării [14]. Este important de menționat că există unele cazuri în care pot apărea coliziuni de hashing, ceea ce poate necesita verificări suplimentare pentru confirmarea potrivirii efective a șablonului în text.

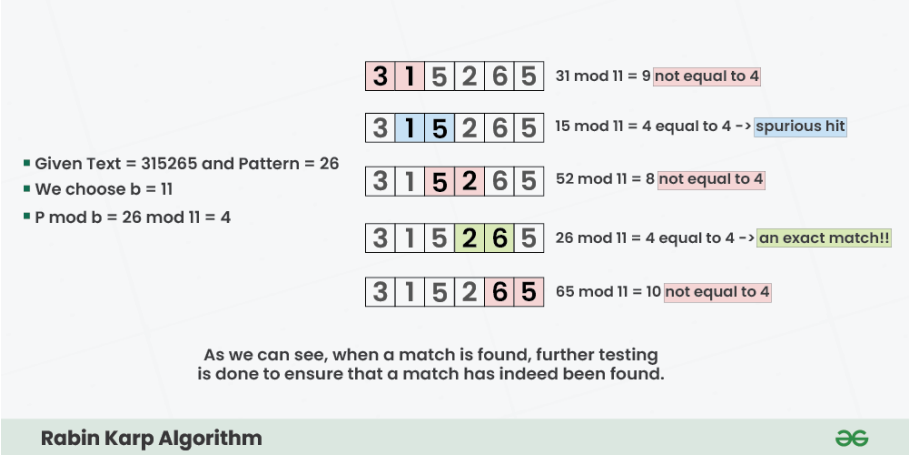


Fig. 5 - Exemplu de Rabin Karp [15]

Un alt algoritm ce caută cuvinte într-un text este FlashText [16]. Conform autorului Vikash Singh, algoritmul are o complexitate de timp liniară, O(n). Algoritmul de față are o serie de particularități întâlnite și în soluția prezentată în această lucrare, precum structura de date de tip arbore de prefixe (trie) și proprietatea că funcția de căutare nu returnează rezultate parțiale.

Principala și cea mai importantă diferență dintre cei doi algoritmi și, în fine majoritatea algoritmilor prezentați mai sus, este aceea că, în timp ce aceștia se specializează pe căutarea unei serii de cuvinte într-un text, algoritmul dezvoltat de mine caută o frază, compusă dintr-unul sau mai multe cuvinte, într-o serie de texte. Această diferență majoră schimbă paradigma din "acul în carul cu fân" în "trusa de cusut în carele cu fân", lucru care înseamnă oportunitatea de a lărgi atât domeniul datelor de intrare, cât și dimensiunea setului de date cunoscute.

Desigur că, precum și autorul menționează, FlashText nu depinde efectiv de numărul de cuvinte căutate, în timp ce algoritmul meu de căutare depinde în totalitate de această variabilă, făcându-l, în unele cazuri, mai lent.

Totuși, am ales să pun accent pe aplicabilitatea algoritmului și să renunț la o parte din performanță. Acceptând intrări cu o entropie mai mare de la utilizator, am extins aria în care algoritmul este relevant. Conform studiului [17], numărul mediu de cuvinte dintr-o căutare pe Google este între 3 și 4, ajungând la concluzia că, din punct de vedere statistic, numărul de cuvinte căutate într-o frază poate fi relativ mic și, deci, poate fi limitat de către aplicațiile ce folosesc o astfel de căutare pentru viteze mai mari. Pentru aplicația dezvoltată de către

mine ce pune în folosință acest algoritm, am ales să limitez numărul de cuvinte căutate într-o frază la 10. Voi descrie în detaliu performanța algoritmului și factorii ce îi influențează viteza în capitolul aferent acestuia.

Un alt atuu este și acela că textele în care voi căuta fraza sunt distribuite în mai multe fișiere și, deși putem ajunge prin concatenare la un text care să le includă pe toate, nu este eficient ca acest text să fie ținut în memorie în tot timpul rulării programului. De asemenea, implementarea conține în rezultatele căutării nu doar răspunsul Am găsit/Nu am găsit, ci și documentul și pagina unde se află informația respectivă.

Capitolul 3

Structura aplicației și tehnologiile utilizate

3.1 Structura aplicației

Acest capitol are rolul de a prezenta structura și arhitectura aplicației noastre, precum și tehnologiile utilizate în dezvoltarea ei. Vom discuta despre modul în care am împărțit aplicația în servicii separate, fiecare având un rol bine definit în cadrul sistemului, și despre modul în care aceste servicii interacționează între ele pentru a oferi funcționalitatea dorită.

Am ales această abordare a multiplelor microservicii în mare măsură datorită următorului principiu teoretic: în cazul în care unul dintre microservicii nu merge, restul aplicației va funcționa în continuare. Dacă serviciul de procesare a imaginilor nu funcționează, căutarea din documente mai vechi poate fi încă folosită de către utilizator și, de asemenea, dacă serviciul de căutare/indexare nu este disponibil, documentele pot fi încărcate și trimise către procesare.

De asemenea, prin utilizarea microserviciilor, gestionarea actualizărilor și implementarea de noi funcționalități devine mai ușoară și mai organizată. Fiecare microserviciu poate fi dezvoltat, testat și implementat independent, fără a afecta celelalte părți ale aplicației. Totodată, în cadrul unei arhitecturi bazate pe microservicii, este mai ușor să se adopte tehnologii și limbaje de programare diferite pentru fiecare serviciu, în funcție de cerințele specifice. Aceasta oferă flexibilitate și libertate în alegerea celor mai potrivite tehnologii pentru fiecare componentă a sistemului, contribuind la optimizarea performanței și eficienței globale a aplicației.

Aplicația este împărțită în trei microservicii, fiecare având rolul său în buna funcționare a aplicației.

3.2 Serviciul de procesare a imaginilor

Primul microserviciu este serviciul ce se ocupă cu procesarea imaginilor. Acesta funcționează ca un Worker ce verifică în mod constant baza de date pentru noi intrări ce se află în pasul de procesare. Apoi, când acesta găsește documente în acest pas, va accesa calea de pe disk, va dezarhiva imaginile și le va aplica algoritmul de recunoaștere a caracterelor (OCR) pentru a genera fișierele de tip text aferente fiecărei pagini. Acest serviciu este dezvoltat folosind limbajul Python datorită integrării facile cu algoritmii de OCR și lucrul cu fișiere.

O bună parte din timpul în care am lucrat la această aplicație a reprezentat-o alegerea celui mai bun engine de OCR pentru acest tip de proiect. OCR este o tehnologie care transformă textul de pe imagini sau documente scanate în text editabil și căutabil digital. Această tehnologie este esențială în procesele de digitalizare și automatizare a informațiilor, eliminând necesitatea de a introduce manual datele din documente tipărite în sisteme informatice.

Un aspect important al tehnologiei OCR este capacitatea sa de a recunoaște și de a interpreta caracterele dintr-o imagine sau scanare. Acest lucru se realizează prin analiza pixelilor din imagine și identificarea formelor și a contururilor care reprezintă litere, cifre sau simboluri.

În ultimii ani, tehnologia OCR a avansat semnificativ datorită progreselor în domeniul învățării automate și a rețelelor neuronale. Algoritmii moderni de OCR folosesc tehnici precum rețelele neurale convoluționale (CNN) pentru a realiza recunoașterea caracterelor cu o precizie și o viteză remarcabilă. Aplicațiile tehnologiei OCR sunt diverse și acoperă o gamă largă de domenii. De exemplu, în domeniul afacerilor, OCR este folosit pentru extragerea automată a informațiilor din facturi, chitanțe sau alte documente financiare. În domeniul medical, este utilizat pentru a procesa și a analiza rezultatele testelor și a rapoartelor medicale. În domeniul

securității, OCR este folosit pentru a verifica și a interpreta documente de identitate sau alte documente importante. [18]

3.2.1 Keras OCR, EasyOCR și PYTESSERACT

TensorFlow este o bibliotecă open-source dezvoltată de Google pentru învățarea automată și deep learning. Creată inițial de echipa Google Brain, TensorFlow a devenit rapid una dintre cele mai populare biblioteci de învățare profundă datorită flexibilității și scalabilității sale. Este utilizată atât în cercetare, cât și în industrie pentru a dezvolta și implementa modele complexe de machine learning. [19]

TensorFlow permite dezvoltatorilor să construiască și să antreneze modele de învățare automată de la zero, oferind un grad mare de control asupra arhitecturii și algoritmilor utilizați. De asemenea, suportă construirea de grafuri computaționale dinamice și statice, permițând optimizarea și adaptarea modelului în timpul rulării.

Această bibliotecă este bazată pe dataflow și programarea diferențiabilă și a fost folosită atât pentru cercetare cât și pentru industrie la Google. Atunci când spunem dataflow ne referim la faptul că realizăm calcule adăugând fiecare element într-un graf. Variabilele grafului se numesc "tensori" și operațiile matematice se numesc operatori.

Tensorii sunt vectori multi-dimensionali cu elemente de același tip. Tensorii nu pot fi modificați ceea ce înseamnă că nu putem actualiza conținutul unui tensor, putem doar să creăm un nou tensor.

TensorFlow este conceput pentru a rula pe diverse platforme, de la dispozitive mobile până la clustere de calcul de mare performanță. Suportul său pentru distribuirea antrenamentului pe mai multe GPU-uri și noduri permite antrenarea modelelor mari și complexe pe seturi de date masive.

TensorFlow include un set cuprinzător de instrumente și biblioteci complementare care extind funcționalitățile sale de bază. De exemplu, TensorFlow Lite permite implementarea modelelor pe dispozitive mobile și IoT, în timp ce TensorFlow Extended (TFX) oferă un cadru complet pentru producția modelelor de machine learning.

Cu API-uri de nivel înalt, cum ar fi Keras, TensorFlow facilitează crearea rapidă și simplă a rețelelor neuronale. Keras oferă o abstracție ușor de utilizat pentru construirea și antrenarea modelelor, permițând dezvoltatorilor să se concentreze pe experimentare și inovare.

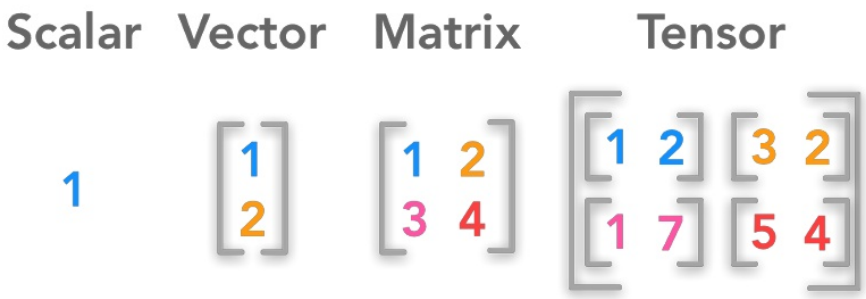


Fig. 9 - Forma unui Tensor [20]

Keras OCR este o bibliotecă Python concepută special pentru sarcinile de recunoaștere optică a caracterelor (OCR). Construită pe baza framework-urilor Keras și TensorFlow, aceasta utilizează capabilitățile puternice de învățare profundă oferite de aceste biblioteci pentru a realiza OCR cu o precizie și eficiență ridicată. Keras OCR este cunoscută pentru API-ul său

user-friendly, ceea ce o face accesibilă dezvoltatorilor și cercetătorilor, chiar și celor care nu au cunoștințe avansate în machine learning sau deep learning. [21]

Keras OCR oferă un API simplu și intuitiv care simplifică procesul de implementare a OCR în aplicațiile Python. Această ușurință în utilizare este unul dintre principalele avantaje ale Keras OCR, permițând dezvoltatorilor să integreze rapid funcționalitatea de OCR fără a fi necesare cunoștințe aprofundate în învățare automată.

Unul dintre marile avantaje ale Keras OCR este disponibilitatea modelelor pre-antrenate. Aceste modele au fost antrenate pe seturi mari de date și pot fi utilizate imediat pentru recunoașterea textului în diverse limbi și stiluri. Această caracteristică reduce semnificativ timpul și efortul necesare pentru configurarea unui sistem eficient de OCR. Acest aspect al bibliotecii Keras a fost și unul dintre principalele motive pentru care am ales-o. Am avut nevoie de aceste modele pre-antrenate pentru a putea eficientiza și accelera faza de dezvoltare a proiectului și ajungând la rezultate optime în cel mai scurt timp.

Nefiind un obiectiv principal al lucrării mele, am ales să folosesc o soluție existentă în locul antrenării modelelor.

Pytorch este un framework open source care a fost folosit pentru a dezvolta unele dintre cele mai faimoase produse care folosesc inteligență artificială. Pytorch a fost derivat din biblionteca Torch bazată pe Lua din 2002 și dezvoltată 14 ani mai târziu, în 2016, în laboratoarele de cercetare ale companiei Facebook. [22]

Pytorch a fost folosit cu succes pentru a antrena modele de computer vision cum ar fi autopilotul de la Tesla, generarea de imagini în proiectul Stable Diffusion și modele de recunoaștere a vorbirii ca Whisper de la OpenAI.

La nivel fundamental, Pytorch este o librărie pentru programarea tensorilor, care sunt, la bază, vectori multi-dimensionali, care vor reprezenta mai târziu date și parametri în rețelele neuronale. PyTorch este cunoscut pentru suportul său pentru grafuri computaționale dinamice. Acest lucru înseamnă că structura grafului poate fi modificată în timpul rulării, făcându-l ideal pentru cercetare și prototipare, unde experimentele și modificările frecvente sunt esențiale.

Datorită accentului pus pe ușurința în utilizare, folosind Pytorch, este posibil ca, doar din câteva linii de cod, să rezulte un model de învățare automată. În plus, el facilitează performanța crescută utilizând calculul paralel pe GPU mulțumită platformei CUDA, dezvoltată de Nvidia. PyTorch oferă suport nativ pentru GPU-uri, permițând accelerarea semnificativă a antrenării modelelor prin utilizarea platformei CUDA dezvoltate de Nvidia. Aceasta face ca antrenarea modelelor mari și complexe să fie mult mai rapidă și mai eficientă.

PyTorch are un ecosistem robust de biblioteci și extensii care completează funcționalitatea sa de bază. De exemplu, TorchVision este o bibliotecă pentru sarcini de viziune computațională, iar PyTorch Lightning oferă un cadru mai structurat pentru scrierea codului PyTorch.

EasyOCR este o bibliotecă open-source pentru recunoașterea optică a caracterelor (OCR) dezvoltată de Jaided AI. Aceasta este cunoscută pentru ușurința în utilizare și capacitatea sa de a recunoaște text în peste 80 de limbi diferite. EasyOCR este construită pe baza framework-ului PyTorch, ceea ce îi conferă avantajele legate de performanță și flexibilitate ale acestuia. [23]

EasyOCR utilizează PyTorch ca backend principal pentru toate operațiunile sale de învățare profundă. Acest lucru înseamnă că modelele de OCR antrenate și utilizate de EasyOCR sunt construite și rulate folosind capabilitățile PyTorch. Alegerea PyTorch se datorează flexibilității și ușurinței de utilizare a acestuia, precum și suportului său robust pentru GPU, care accelerează procesul de antrenare și inferență.

Modelele utilizate în EasyOCR pentru detectarea și recunoașterea textului sunt create

folosind arhitecturi de rețele neuronale care sunt ușor de implementat în PyTorch. Aceste modele beneficiază de dinamismul și flexibilitatea oferite de PyTorch, permițând modificări și optimizări rapide în timpul dezvoltării.

Există multiple motive pentru care EasyOCR este biblioteca de OCR pe care am ales-o pentru proiectul meu, pe lângă rezultatele testelor ce au rezultat într-o mai bună recunoaștere a textului de pe imagini atât clare cât și mai puțin clare.

În primul rând, EasyOCR suportă peste 80 de limbi diferite, ceea ce este esențial pentru proiectele de arhivare digitală care implică documente în multiple limbi. Această capacitate permite digitalizarea corectă și precisă a documentelor internaționale fără a necesita software suplimentar sau personalizări complexe.

De asemenea, la fel cum am specificat anterior, nefiind o funcționalitate pe care se concentrează lucrarea, un alt motiv este acela că biblioteca este concepută pentru a fi simplu de utilizat, având un API intuitiv care permite integrarea rapidă și ușoară în orice aplicație. Pentru un proiect de arhivare digitală, această simplitate reduce timpul și efortul necesare pentru configurare și implementare. Modelele de OCR ale EasyOCR sunt antrenate pe seturi mari de date și sunt capabile să recunoască textul cu o precizie ridicată.

3.2.2 Alte funcționalități

Această aplicație dispune și de un instrument care permite preluarea unui fișier PDF cu conținut text și îl descompune în fișiere individuale de tip text, fără a necesita conversia prin tehnici de recunoaștere optică a caracterelor (OCR). Acest lucru se traduce într-un proces eficient și precis, evitând astfel transformarea textului deja disponibil în format digital în imagini pentru a fi interpretat ulterior. Astfel, prin acest tool, se sare peste pasul de procesare prin OCR și pur și simplu se va prelua acest text dacă în locul unei arhive este prezent un fișier pdf la încărcarea documentului.

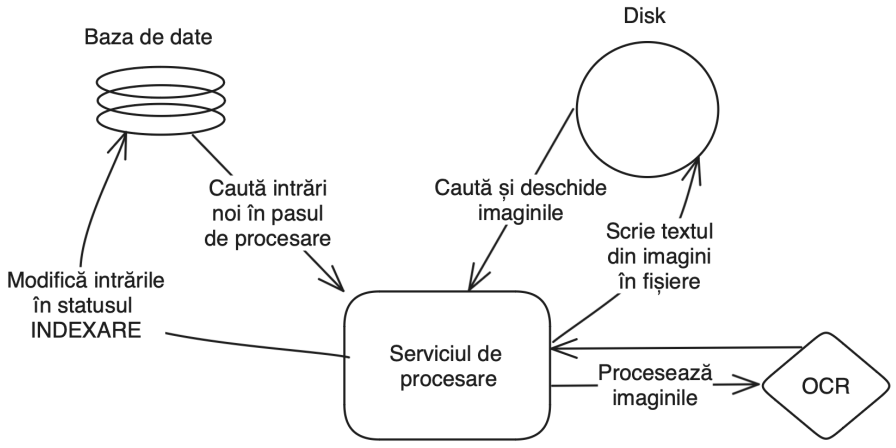


Fig. 6 - Schema legăturilor serviciului de procesare a imaginilor

3.3 Serviciul pentru încărcare și căutare

Serviciul ce se ocupă cu încărcarea documentelor și căutarea informațiilor de către utilizatorul final este un proiect de tip ASP.NET Core Web Application, ce conține atât legătura cu baza de date și proiectul ce se ocupă cu indexarea, cât și interfața cu utilizatorul. Acest serviciu este primul pas în cadrul încărcării unui document în sistem, reprezentând upload-ul efectiv al arhivei în aplicație, copierea acesteia din interfață pe disk și adăugarea intrării în baza de date.

Totodată, acest proiect se ocupă și de căutarea informațiilor în documentele deja procesate și indexate și întoarcerea rezultatelor relevante, alături de documentul și pagina la care se află informația cerută. Legătura cu aplicația de indexare este necesară în momentul căutării, întrucât indexul format se află în memoria programului de indexare.

În momentul instalării sistemului pentru utilizatori, există opțiunea de a configura persistența indexului astfel încât, în cazul în care aplicația de indexare se închide și se redeschide, indexul să fie preluat, dintr-un fișier json sau dintr-o bază de date de tip document (mongo), și nu mai este nevoie de recalcularea acestuia (în cazul unui set mare de date, recalcularea poate dura foarte mult). Aceasta este o optimizare necesară care poate face o diferență foarte mare în disponibilitatea aplicației.

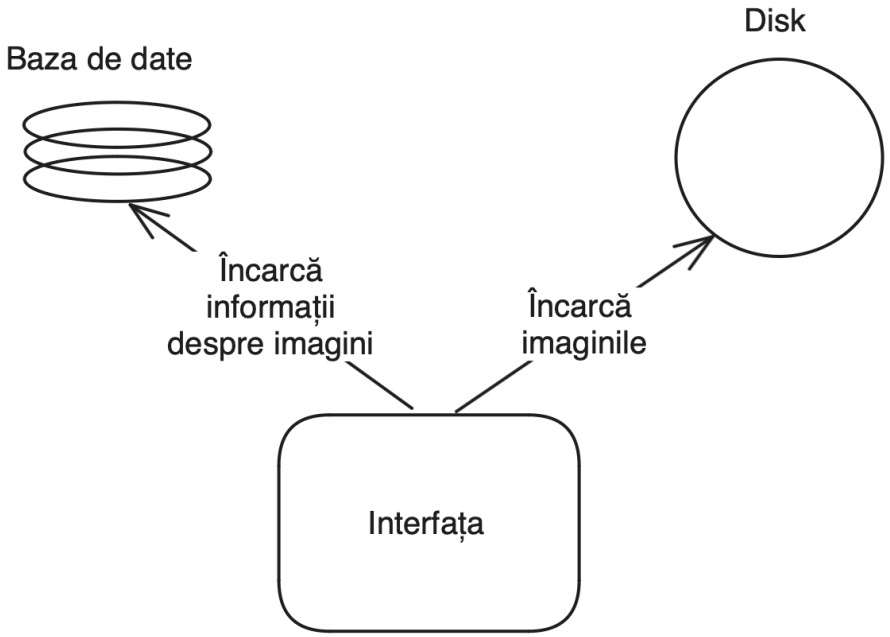


Fig. 8 - Schema legăturilor serviciului de interfață

3.4 Serviciul de indexare

Serviciul de indexare este o combinație de două tipuri de proiecte: un Worker ce verifică în mod constant baza de date pentru intrări noi ce se află în pasul de indexare și un API ce asigură legătura cu interfața pentru a îi transmite rezultate ale căutării. Serviciul conține o clasă de bază de tip Singleton, mai specific un administrator de operații efectuate asupra indexului, ce se asigură că informațiile sunt actualizate constant și la timp. Indexul este stocat în memorie sub forma unui dicționar, cheia fiind un cuvânt, iar valoarea un arbore de prefixe (trie) cu următoarele L cuvinte de după el. În funcție de preferințele utilizatorului, această structură de date este fie persistentă și se stochează sub forma unui fișier json, fie se recalculează de fiecare dată când aplicația este pornită.

Comunicarea între cele două servicii se realizează HTTP REST, prin intermediul API-ului oferit de către serviciul de indexare. În acest mod, m-am asigurat că interfața cu utilizatorul poate fi implementată sub orice altă formă, fie o aplicație web cu Angular, React și altele, fie o aplicație de mobil sau chiar una de Desktop. Deoarece întreaga funcționalitate de căutare se poate integra doar folosind API-ul, împărțirea aplicației în componente independente și scalabile a fost efectuată eficient.

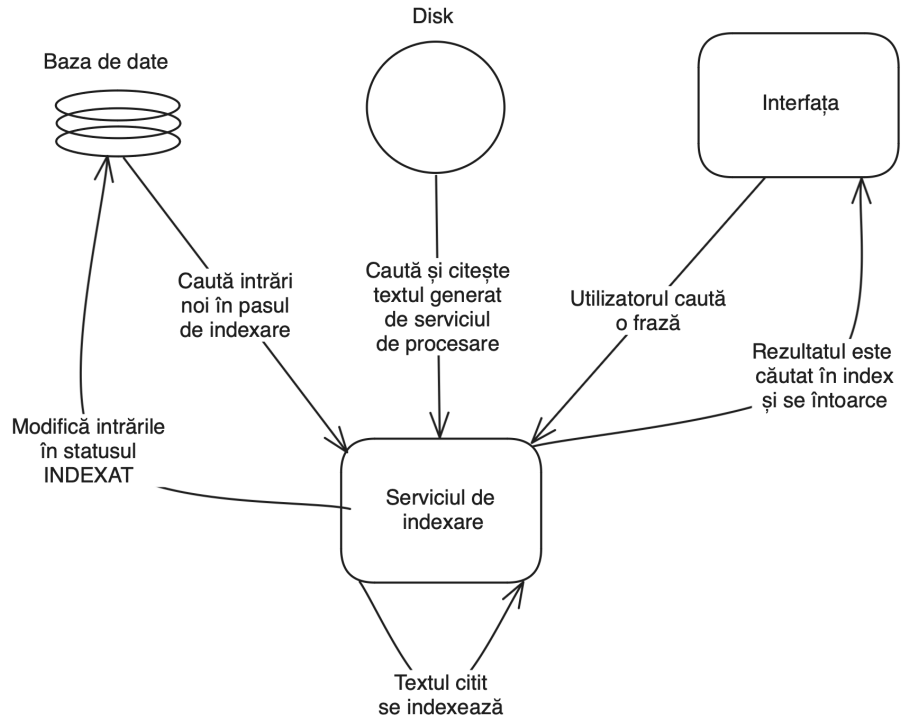


Fig. 9 - Schema legăturilor serviciului de indexare

Capitolul 4

Teste pentru compararea și măsurarea performanței

4.1 Descriere

Pentru a putea vedea dacă algoritmul se poate aplica în cazuri concrete, am creat multiple medii pentru rularea testelor. Presupunem că avem unul sau mai multe documente, fiecare cu mai multe pagini în format text, acestea fiind rezultatul aplicării pasului de procesare a imaginilor. Toate aceste fișiere vor fi stocate în memorie, unele ca și tip de date string, celelalte folosind structura de date pentru indexare.

Acestor date de intrare li se vor aplica algoritmii de căutare ce au fost descriși în capitolul "Metode existente" și se va compara performanța în ce privește timpul de căutare. Am dezvoltat și o serie de teste ce măsoară performanța din punct de vedere al preprocesărilor necesare, unde este cazul.

4.2 Configurare

Toți algoritmii au fost scriși în framework-ul .NET Core 8.0 și C# 12.0 și pentru măsurarea performanței a fost folosită biblioteca BenchmarkDotNet [24]. În prima parte a testării, biblioteca are nevoie de un set de date de intrare ce vor fi schimbate pe parcursul rulării testelor și de definirea funcțiilor ce se vor ocupa de pregătirea mediului pentru rulare. În cazul algoritmului meu, funcția "Setup" va construi calea către folderul în care se află toate documentele și va construi indexul propriu zis luând fiecare fișier (pagină) din fiecare document în parte, citindu-i conținutul și aplicând algoritmul descris anterior.

```
[GlobalSetup(Target = nameof(SearchPhrase))]
public void Setup()
{
    var documentsPath = "D:\\Work\\file-organiser\\BenchMarkApp\\BenchMarkApp\\Files\

    var documents = Directory.GetDirectories(documentsPath);

    foreach (var documentName in documents)
    {
        IndexDocument(documentName, documentName);
    }
}
```

Fig. n - Funcția "Setup" pentru indexare

Restul algoritmilor vor avea câte o funcție de "Setup" în care, asemănător cu metoda de indexare, va citi conținutul tuturor documentelor dar le va stoca într-o listă cu elemente de tip șir de caractere și va instanția clasa aferentă fiecăruia.

```
public Dictionary<string, List<string>> TextFiles;

private BoyerMoore BoyerMoore;
private RabinKarp RabinKarp;

[GlobalSetup(Targets = new[]
{
```

```
    nameof(SearchPhraseNaive),
    nameof(SearchPhraseRegex),
    nameof(SearchBoyerMoore),
    nameof(SearchRabinKarp),
    nameof(SearchKMP)
})]
public void SetupOthers()
{
    BoyerMoore = new BoyerMoore(Encoding.ASCII.GetBytes(SearchedString));
    RabinKarp = new RabinKarp();
    TextFiles = new Dictionary<string, List<string>>();
    var documentsPath = "D:\\Work\\file-organiser\\BenchMarkApp\\BenchMarkApp\\Files\
    var documents = Directory.GetDirectories(documentsPath);
    foreach (var documentName in documents)
    {
        var files = Directory.GetFiles(documentName);
        foreach (var filePath in files)
        {
            if (Path.GetExtension(filePath) == ".zip" || Path.GetExtension(filePath) =
            {
                continue;
            }
            var text = GetTextFromFile(filePath);
            if (!TextFiles.TryGetValue(documentName, out var textFiles))
            {
                textFiles = new List<string>();
                TextFiles.Add(documentName, textFiles);
            }
            textFiles.Add(text);
        }
    }
}
```

Fig. 10 - Funcția "Setup" pentru restul algoritmilor de căutare

Acest benchmark creat va fi rulat și, după ce toate testele configurate vor fi finalizate, va afișa un tabel de metrici, fiecare intrare reprezentând tipul algoritmului folosit dar și parametrii cu care s-a realizat testul.

Din punctul de vedere al dimensiunii setului de date de testare, s-au folosit variabile pentru a determina viteza de rulare a algoritmilor atât pentru un document cât și pentru 10 documente, fiecare având 15 pagini. Acest test va determina relevanța și aplicabilitatea algoritmilor în diverse situații, precum utilizarea sa într-un singur fișier text sau pentru o întreagă carte sau colecții de cărți.

Paginile au fost extrase din multiple cărți din literatura clasică, toate în format pdf cu conținut de tip text. Pentru a extrage conținutul din aceste pdf-uri exact în formatul dorit, mai exact un fișier text pentru fiecare pagină, am folosit tool-ul dezvoltat de mine în cadrul aplicației de Python ce primește ca date de intrare un fișier pdf și extrage toate paginile în forma descrisă anterior. Acest tool poate fi folosit fie separat, fie împreună cu aplicația principală caz în care se sare peste pasul de OCR.

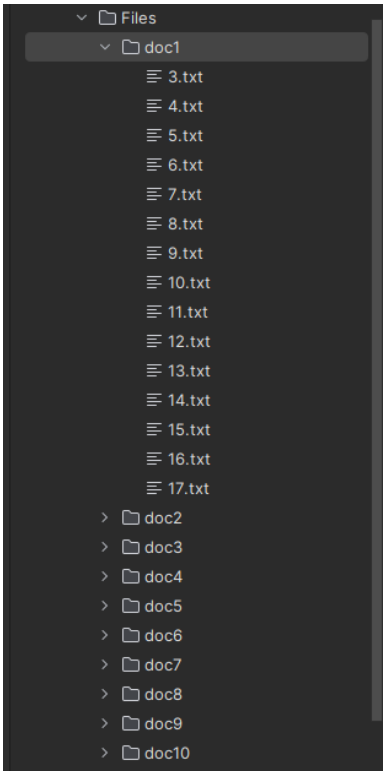


Fig. 11 - Structura documentelor de test

În final, după rularea testelor de căutare au rezultat tabele ce măsoară performanțele algoritmilor, concluzionând clar că algoritmul meu este mult mai rapid comparativ cu ceilalți. Putem observa faptul că, în timp ce timpul de rulare pentru restul algoritmilor crește exponențial în funcție de numărul documentelor și al numărului de cuvinte al frazei căutate, timpul de rulare al algoritmului meu crește relativ puțin de la test la test. Acești timpi de rulare se modifică relativ la dimensiunea textului căutat și poziția sa în textul în care se caută.

```
public static IEnumerable<string> ValuesForSearchedString => new[]
{
    "A joy that",
    "But this is quite",
    "And no work they should be doing But watching their"
};
```

Fig. 12 - Valorile textului căutat

Method	SearchedString	Mean	Error	StdDev	Median
SearchPhrase	A joy that	157.0 ns	1.81 ns	1.51 ns	156.9 ns
SearchPhraseNaive	A joy that	191,710.5 ns	458.59 ns	382.94 ns	191,619.1 ns
SearchPhraseRegex	A joy that	18,057.7 ns	151.70 ns	141.90 ns	18,062.1 ns
SearchBoyerMoore	A joy that	120,408.5 ns	928.45 ns	823.04 ns	120,294.6 ns
SearchRabinKarp	A joy that	520,618.4 ns	347.66 ns	308.19 ns	520,507.7 ns
SearchKMP	A joy that	297,975.4 ns	5,821.48 ns	10,195.85 ns	303,114.8 ns
SearchPhrase	And n(...)their [51]	386.4 ns	2.65 ns	2.48 ns	385.7 ns
SearchPhraseNaive	And n(...)their [51]	194,443.0 ns	1,270.76 ns	1,188.67 ns	194,672.4 ns
SearchPhraseRegex	And n(...)their [51]	21,279.0 ns	88.02 ns	82.33 ns	21,278.7 ns
SearchBoyerMoore	And n(...)their [51]	67,922.8 ns	808.86 ns	756.61 ns	68,094.7 ns
SearchRabinKarp	And n(...)their [51]	717,178.1 ns	281.43 ns	235.01 ns	717,079.3 ns
SearchKMP	And n(...)their [51]	323,621.1 ns	1,097.14 ns	1,026.27 ns	323,830.9 ns
SearchPhrase	But this is quite	195.7 ns	0.63 ns	0.52 ns	195.7 ns
SearchPhraseNaive	But this is quite	147,276.1 ns	406.35 ns	380.10 ns	147,295.8 ns
SearchPhraseRegex	But this is quite	14,930.4 ns	60.99 ns	54.06 ns	14,924.8 ns
SearchBoyerMoore	But this is quite	92,239.2 ns	1,610.59 ns	1,427.75 ns	91,908.6 ns
SearchRabinKarp	But this is quite	522,554.2 ns	266.20 ns	207.83 ns	522,526.4 ns
SearchKMP	But this is quite	320,750.6 ns	5,573.41 ns	5,963.49 ns	320,317.8 ns

Fig. 13 - Tabelul rezultat din rularea testelor generat de biblioteca BenchmarkDotNet

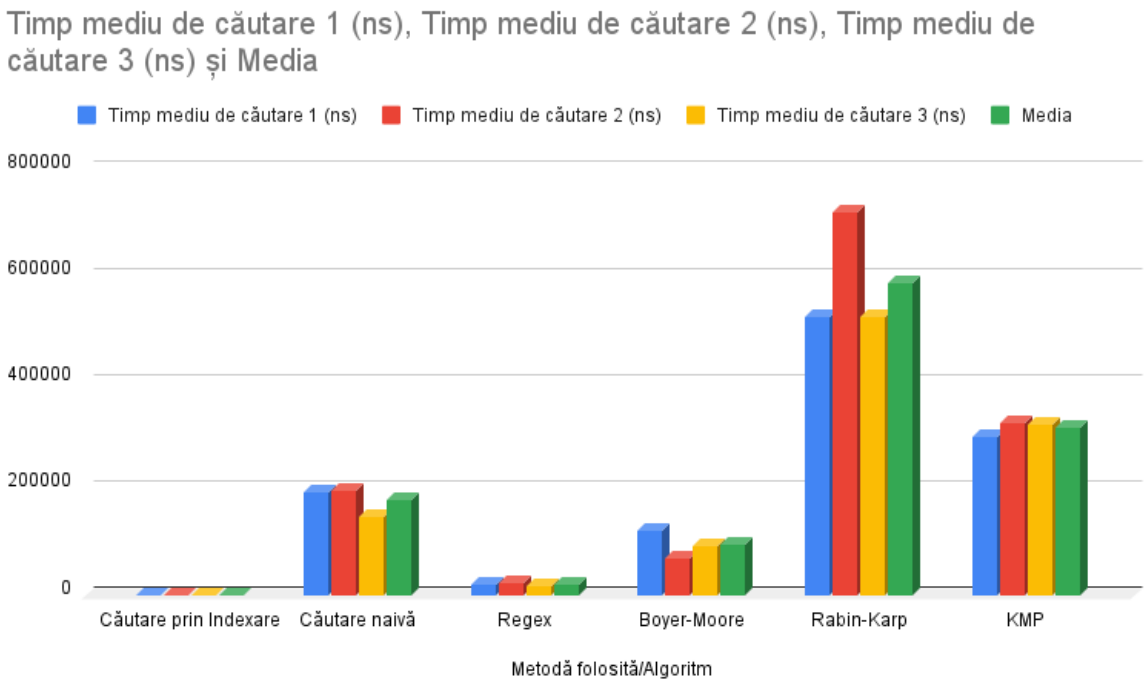


Fig. 14 - Graficul ce pune în perspectivă performanțele algoritmilor

Capitolul 5

Concepte teoretice

5.1 Descriere

Fiind o lucrare în special ce se bazează pe partea teoretică în principal și pe demonstrarea alegerilor făcute în implementarea propriu-zisă, am făcut o serie de documentări în ce privește noțiunea de complexitate a algoritmilor și diverse structuri de date ce permit deblocarea unor îmbunătățiri ce pot fi aduse căutării de text. Conceptele ce urmează a fi prezentate fie au jucat un rol în implementarea algoritmilor de indexare și căutare, fie au adus un aport teoretic de bază de la care am putut să plec cu scopul adăugării componentelor necesare.

5.2 Paradigma indexării în contextul curent

Pornind de la motivele invocate în capitolele anterioare, am concluzionat că memoria ocupată de un program ce se ocupă cu căutarea unui subtext într-un text mai mare poate fi, într-o oarecare măsură, neglijată. Deși, la prima vedere, un algoritm ce trebuie să itereze prin mai multe serii de texte ar trebui să folosească mai intens procesorul decât memoria, conceptul de indexare redefinește paradigma cu totul.

Indexarea reprezintă procesul de creare a unei structuri de date optimizate pentru căutarea eficientă a informațiilor într-o bază de date sau într-un set de date mai larg. În contextul motoarelor de căutare, indexarea reprezintă procesul esențial prin care informațiile din întregul internet sunt organizate, structurate și pregătite pentru a fi accesate rapid și eficient de către utilizatori. Acest proces are o importanță crucială în funcționarea motoarelor de căutare, cum ar fi Google, Bing sau DuckDuckGo, deoarece determină capacitatea acestora de a oferi rezultate relevante și actualizate în timp real. [25]

Indexarea începe cu "crawling-ul", adică parcurgerea paginilor web de către roboți de căutare sau crawlers. Acești roboți explorează constant internetul, descoperind și actualizând informațiile din diverse pagini web și urmăresc link-urile dintr-o pagină web către alte pagini web și colectează informații relevante. Crawlers încep de obicei de la un set de pagini web cunoscute (denumite și seed-uri) și urmăresc apoi linkurile pentru a descoperi alte pagini. Acest proces este continuu și dinamic, deoarece internetul se schimbă constant, iar paginile noi sunt create sau actualizate în mod regulat. Crawlers colectează diverse tipuri de date despre paginile web, cum ar fi URL-urile, conținutul textului, titlurile, descrierile, imagini, linkurile, structura paginii etc. Aceste informații sunt apoi procesate și adăugate în indexul motoarelor de căutare pentru a fi accesate rapid în timpul căutărilor. Crawlers respectă protocoalele web, cum ar fi robots.txt și meta robots, care indică paginilor ce trebuie sau nu trebuie parcurse de către roboți. Aceste protocoale sunt esențiale pentru respectarea politicilor de confidențialitate și securitate ale site-urilor web.

Aceste informații sunt apoi adăugate într-o bază de date specială numită index. Indexul este esențial pentru căutare deoarece optimizează căutările ulterioare. În loc să parcurgă repetat toate paginile web de fiecare dată când se face o căutare, motorul de căutare se referă la index pentru a returna rezultate relevante și actualizate în funcție de termenii căutați și de algoritmii de ranking utilizați.

Principalele funcții ale indexului sunt de a accelera căutările și a le grupa. În loc să parcurgă toate înregistrările sau toate paginile pentru a găsi informația căutată, sistemul de căutare se referă la index pentru a identifica rapid locația și informațiile relevante, ceea ce reduce semnificativ timpul de căutare și procesare a datelor. Indexul permite implementarea căutărilor complexe și a filtrărilor bazate pe diferite criterii, cum ar fi cuvinte cheie, categorii, date, autor etc.

Procesul de căutare și clasificare (sau ranking) reprezintă etapa finală în funcționarea unui motor de căutare. După ce informațiile au fost colectate și indexate în etapele anterioare (crawling și indexing), utilizatorul poate să efectueze căutări și să primească rezultate relevante în funcție de cererea sa. Procesul începe atunci când utilizatorul introduce un query sau o căutare în motorul de căutare. Acest query poate fi un cuvânt cheie, o frază sau o întrebare complexă. Motorul de căutare interpretează această căutare pentru a înțelege intenția utilizatorului și a returna cele mai relevante rezultate. După ce query-ul este interpretat, motorul de căutare accesează indexul său pentru a recupera informațiile relevante. Această etapă implică aplicarea unor algoritmi de căutare și filtrare pentru a selecta rezultatele care corespund cel mai bine cererii utilizatorului. Motoarele de căutare se concentrează și pe îmbunătățirea experienței utilizatorului prin oferirea unor rezultate relevante, rapide și ușor de accesat. Acest lucru poate implica personalizarea rezultatelor în funcție de istoricul căutărilor utilizatorului, oferirea sugestiilor sau completărilor automate la căutările în timp real, și alte funcționalități care îmbunătățesc utilizarea motorului de căutare. [26]

Datorită faptului că se cunoaște modul în care acești algoritmi complecși de indexare funcționează, s-au găsit metode prin care pot fi „păcăliți". Tehnica SEO (Search Engine Optimization) reprezintă un set de practici și strategii utilizate pentru a îmbunătăți vizibilitatea și clasamentul unui site web în rezultatele motoarelor de căutare. Aceste practici sunt dezvoltate pentru a se alinia cu criteriile de indexare și clasificare ale motoarelor de căutare, astfel încât să îmbunătățească poziția unui site în rezultatele căutării și să atragă mai mulți vizitatori organici. Există mai multe tehnici de SEO care pot fi utilizate pentru a optimiza un site web și a-l face mai "atractiv" pentru motoarele de căutare, cum ar fi Google. Câteva dintre aceste tehnici sunt: optimizare pentru cuvinte cheie, optimizarea conținutului, optimizare tehnică, link building, etc. [27]

5.3 Structuri de date

În cadrul algoritmului de indexare au fost folosite multiple structuri de date, fiecare având rolul său. Acestea sunt interconectate și formează o structură de date complexă ce reține informațiile despre toate documentele aplicației.

Clasa denumită IndexDict, care este folosită pentru a defini obiectul ce reține informațiile efective, urmărește design pattern-ul Singleton. Un Singleton este un design pattern de creare a obiectelor în programare orientată pe obiecte, care se concentrează pe faptul că o clasă trebuie să aibă o singură instanță și să ofere un mod global de a accesa această instanță [28]. Am ales să utilizez acest pattern deoarece indexul este unic în această instanță a aplicației și nu se va distruge sau creea un altul în timpul rulării.

IndexDict conține un dicționar, o structură de date care stochează date într-o asociere cheie-valoare, oferind o modalitate eficientă de căutare, adăugare și eliminare a elementelor folosind o cheie unică asociată fiecărui element. Elementele sunt stocate sub formă de perechi cheie-valoare, unde cheia este un identificator unic care permite accesul rapid la valoarea asociată. Dicționarele sunt optimizate pentru căutare, folosind structuri de date interne precum tabele de dispersie sau arbori, pentru acces rapid la elemente în funcție de cheie. Acestea sunt flexibile în privința tipurilor de date pe care le pot stoca și necesită chei unice; astfel, adăugarea unei chei existente înlocuiește valoarea asociată cu noua valoare. [29] Cheile acestui dicționar sunt reprezentate de toate cuvintele unice din toate documentele procesate, iar valoarea este reprezentată de un arbore de sufixe (Trie) ce conține cuvântul din cheie

ca și rădăcină, iar copiii săi sunt noduri ce conțin cuvintele ce urmează după acesta într-o înșiruire logică (în cazul algoritmului meu, înșiruirea logică este o propoziție ce se termină cu un semn de punctuație - ".!? \n"). În mod evident, propozițiile sunt prezente în textul în care se dorește a fi realizată căutarea, mai exact în fișierele ce conțin textul aferent unei pagini din document. De asemenea, clasa conține și o constantă de tip întreg denumită "MaxIndexCount" ce stochează numărul maxim de cuvinte ce pot fi stocate în arborele de sufixe, reprezentând totodată și adâncimea arborelui. Valoarea acestei variabile devine relevantă în cazul în care propozițiile sunt relativ lungi iar stocarea lor în arbore nu ar mai fi eficientă sau, în cazul în care textul nu are semne de punctuație, arborele ar stoca întregul conținut al uneia sau mai multor pagini. Această variabilă dictează și numărul maxim de cuvinte ce pot fi căutate de către utilizator din interfață și poate fi configurată în momentul instalării aplicației.

Metodele clasei "IndexDict" sunt folosite pentru a aduce schimbări valorilor din index și pentru a aplica acțiuni de căutare în acest index. Putem spune că, din punct de vedere teoretic, această clasă are un nivel de abstractizare mai scăzut și poate fi clasificată drept o clasă driver ce face legătura dintre dicționarul ce conține informațiile indexului și metodele deschise către interfața cu utilizatorul.

Metoda "AddItem" este folosită pentru a adăuga un cuvânt și lista de cuvinte următoare în arborele de indexare. Dacă cuvântul nu există în index, se creează un nou arbore pentru acel cuvânt și se adaugă lista de cuvinte următoare. Dacă cuvântul există deja, se adaugă doar lista de cuvinte următoare în arborele existent. "ProcessText" primește un text și numele documentului și procesează textul pentru a-l împărți în cuvinte. Pentru fiecare cuvânt, adaugă cuvintele următoare în arborele de indexare, limitându-se la o anumită adâncime dată de constanta "MaxIndexCount". Metoda "SearchPhrase" primește o listă de cuvinte care reprezintă fraza din interogarea utilizatorului și caută acea frază în index. Verifică dacă primul cuvânt din frază există în index și, dacă da, navighează prin arborele asociat pentru a verifica dacă celelalte cuvinte din frază sunt prezente în aceeași ordine în arbore.

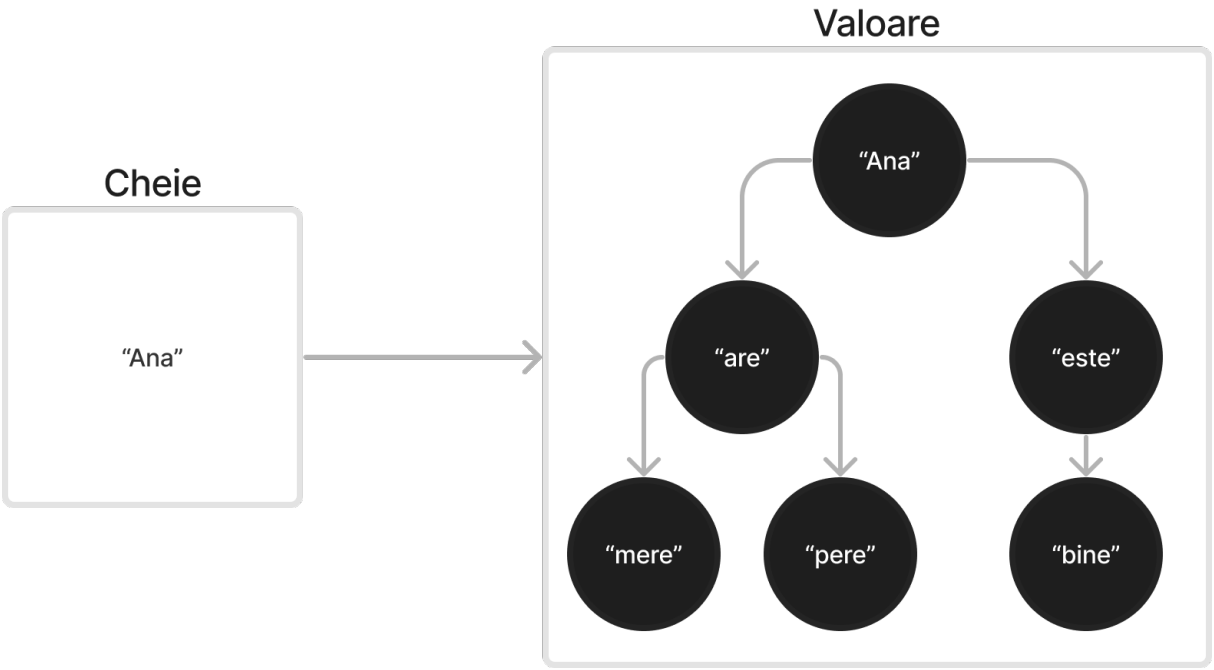


Fig. n - Exemplu de intrare în dicționarul ce stochează indexul

Clasa "Tree" este o clasă cu o singură proprietate și anume "Root", ce reține rădăcina

arborelui. Acest arbore este implementat sub formă recursivă, fiecare nod din arbore având o listă cu toți copiii săi (în cazul acestei implementări, pentru o eficientizare mai bună a căutării, lista de copii este tot un dicționar de tip cuvânt -¿ nod). Acest arbore nu este doar un arbore oarecare ci este interpretat ca un Trie, sau arbore de sufixe. Într-un Trie, fiecare nod reprezintă un caracter sau o secvență de caractere dintr-un cuvânt sau text. Astfel, navigarea în acest arbore se face pe baza caracterelor, ceea ce îl face ideal pentru căutarea rapidă a cuvintelor într-un text. Un alt avantaj al utilizării unui Trie constă în faptul că, deși poate necesita mai multă memorie decât alte structuri de date pentru stocarea cuvintelor, acest aspect este compensat de eficiența sa în căutare. Prin interpretarea arborelui ca un Trie în contextul indexării și căutării cuvintelor în text, se obține o structură de date optimizată pentru procesele de căutare, contribuind la realizarea operațiilor de indexare și căutare într-un mod eficient și rapid. [30]

În clasa "Tree" există o singură metodă care se numește "AddListOfWords" și primește ca parametrii o listă de cuvinte și un obiect de tip "DocumentPage" ce conține informații despre numele documentului și pagina în care se află cuvântul care se adaugă. Funcția operează în felul următor: iterăm prin fiecare cuvânt din lista de cuvinte dată, dacă nodul curent are un copil cu valoarea egală cu cuvântul curent atunci se trece la nodul respectiv și, simultan, la următorul cuvânt. Dacă nu există un nod copil ce are valoarea egală cu acest cuvânt, se creează un nod nou cu această valoarea și se va trece la el ca fiind nodul curent. În acest fel, în cazul unei fraze noi, toate cuvintele vor fi adăugate ca și noduri noi. Logica de adăugare de nod copil se află în clasa "TreeNode" despre care voi vorbi în rândurile ce urmează.

Motivația din spatele alegerii acestei structuri de date este dată de faptul că este una dintre structurile de date ce pot cel mai bine să stocheze o mulțime dinamică de șiruri, în special șiruri de caractere. În cazul algoritmului de indexare, în acest Tree nu se vor stoca în noduri caracterele cuvintelor, deoarece acest proces ar fi foarte ineficient și într-o oarecare măsură redundant, ci se vor stoca cuvintele. Nodurile frunză ale arborelui reprezintă finalul propoziției sau frazei din text, iar, spre deosebire de implementarea clasică cu caractere, vom ști că oriunde în parcurgerea în adâncime a acestui arbore avem o propoziție sau frază ce există în textul inițial, nefiind necesară parcurgerea până la ultimul nivel. Această proprietate ajută la întoarcerea rezultatelor din potriviri parțiale ale frazelor dar potriviri exacte ale cuvintelor.

Aici scrie despre DFS [31]

Fig. n - Modul în care cuvintele sunt căutate în Trie

Explicație: Cuvintele "Ana", "are", "mere", "și", "pere" sunt noduri în Trie iar verificarea se face pe existența cuvintelor întregi și nu se potrivesc parțial, precum este perechea "mere" și "me". În cel de-al doilea caz, deși cuvântul "pere" se află într-un nod din copiii cuvântului "Ana", nu se află direct în continuarea cuvântului "are" și, deci, nu este o potrivire.

Fiecare nod din Trie este o instanță a clasei "TreeNode", ce stochează valoarea cuvântului de la nivelul respectiv (string), un dicționar de tip cuvânt - nod, ce conține toți copiii săi și un HashSet de obiecte de tip "DocumentPage". Structura de date de tip HashSet în limbajul C# reprezintă o colecție neordonată de elemente unice, bazată pe utilizarea unui tabel de dispersie (hash table). Această implementare este eficientă pentru operațiile de adăugare, căutare și eliminare a elementelor, având o complexitate medie de O(1) pentru majoritatea operațiilor, în cazul unei bune funcționări a funcției de hash. HashSet este optimizat pentru acces rapid la elementele unice și nu permite duplicatele în colecție. Atunci când se adaugă un element nou în HashSet, acesta este adăugat doar dacă nu există deja în colecție. Acest aspect face HashSet-ul potrivit pentru stocarea și gestionarea unor mulțimi de date distincte, cum ar fi în cazul indexării de cuvinte într-un text sau în gestionarea unor liste de elemente unice într-o aplicație. Am ales folosirea acestui tip de stocare a unei mulțimi de elemente deoarece, în primul rând, am nevoie ca intrările din aceeași pagină a aceluiași document să nu apară de mai multe ori, fiind o informație redundantă, și apoi, datorită eficienței sale în întoarcerea de valori, poate fi folosită foarte eficient pentru filtrarea rezultatelor după un anumit document sau pagină. [32]

"TreeNode" conține metode ce operează cu nodul curent și copiii acestuia, în speță "AddChild" pentru adăugarea unui copil cu o anumită valoare și un obiect de tip "DocumentPage" și operații de scriere, citire, ștergere, actualizare și curățare a nodului curent. Aceste operații se folosesc în principal de viteza cu care se lucrează pe un tip de date ca HashSet, având complexitatea de timp în medie O(1).

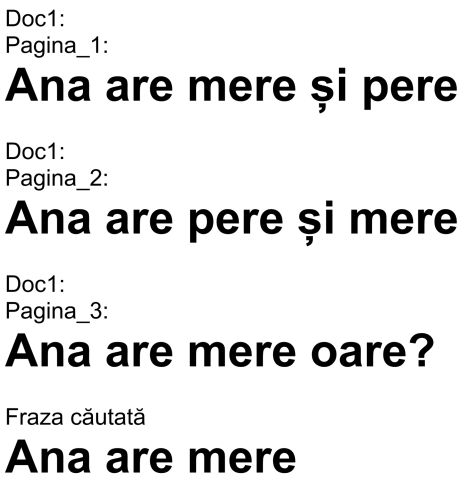
Indexarea propriu-zisă începe în funcția "CheckIndexAsync" din IndexingOperationsManager ce va cere din baza de date toate intrările ce au statusul "Indexing" sau, dacă setarea

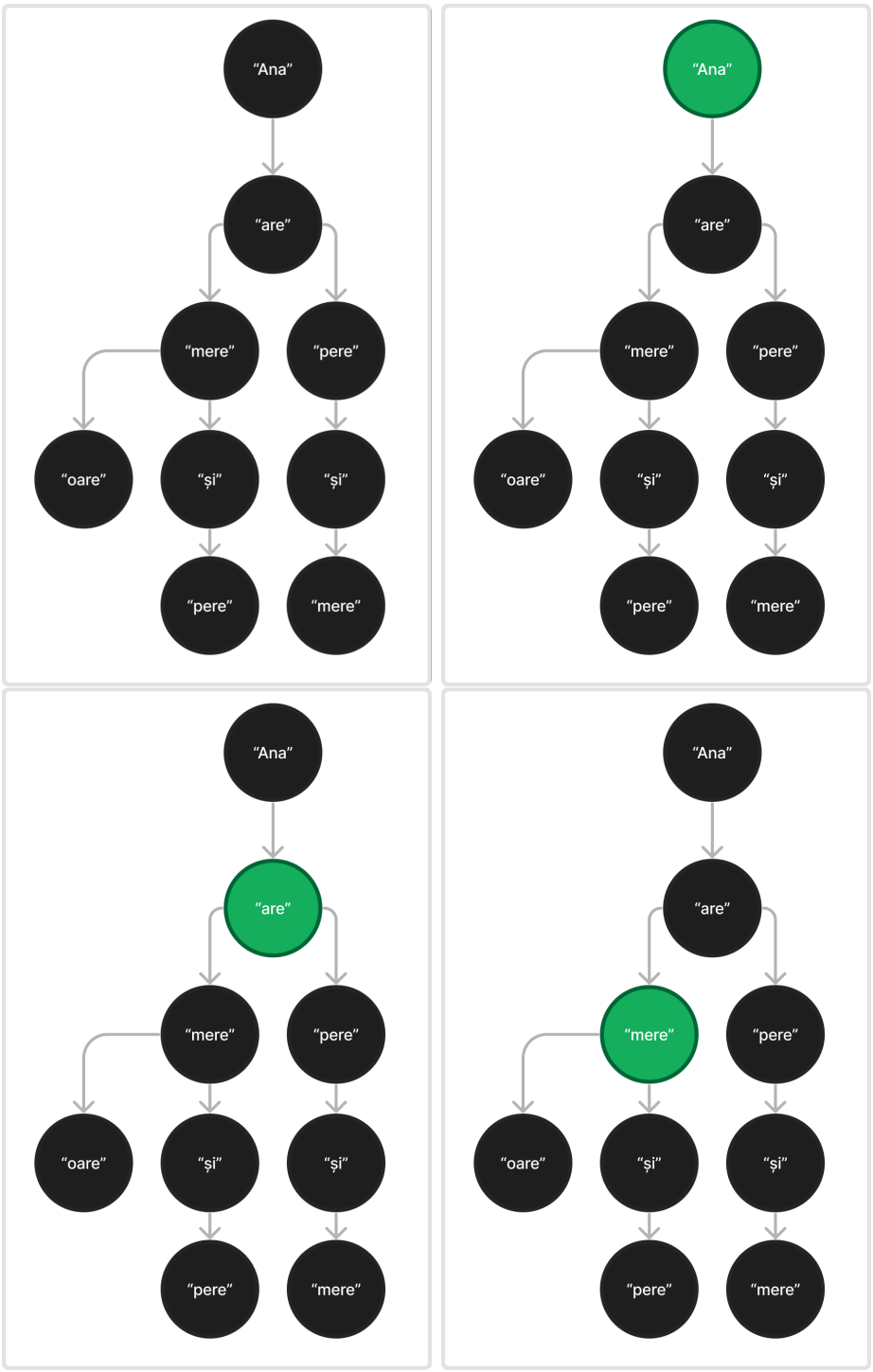
pentru recalcularea indexului există, va prelua toate intrările din baza de date ce au un status mai mare sau egal decât "Indexing". Apoi, se va chema "IndexText" ce cheamă la rândul său metoda "ProcessText" din clasa "IndexDict". Această metodă primește ca parametrii textul ce se dorește a fi indexat, numele documentului și numărul paginii și, folosind regula de împărțire în înșiruiri logice prin care se vor extrage cuvintele, se va adăuga în dicționar fiecare cuvânt cu arborele asociat acestuia, format din restul cuvintelor din propoziție sau numărul maxim de cuvinte.

Metoda ce se ocupă de căutare primește ca parametru o listă de cuvinte reprezentând o frază sau un șir de cuvinte pe care utilizatorul dorește să le caute în text. Procesul de căutare începe prin verificarea dacă primul cuvânt din frază există în index. Dacă acesta nu există, se consideră că fraza nu a fost găsită și se returnează un rezultat cu proprietatea "IsFound" setată pe false.

În cazul în care primul cuvânt din frază este găsit în index, se navighează în arborele Trie corespunzător acelui cuvânt și se verifică dacă următoarele cuvinte din frază există în continuare în arbore, în ordinea specificată de frază. Procesul de navigare continuă până când se ajunge la finalul frazei sau când se întâlnește un cuvânt care nu are un copil corespunzător în arbore.

Dacă s-a ajuns la finalul frazei în arbore, acest lucru indică faptul că fraza completă a fost găsită în text și rezultatul căutării este setat ca fiind găsit ("IsFound" = true). În acest caz, se inițializează o listă de pagini de document asociate cu fraza găsită pentru a se adăuga rezultatului alături de proprietatea "IsFound".




"Ana"
"are"
"mere"
"pere"
"oare"
"și"
"și"
"pere"
"mere"
"Ana"
"are"
"mere"
"pere"
"oare"
"și"
"și"
"pere"
"mere"
"Ana"
"are"
"mere"
"pere"
"oare"
"și"
"și"
"pere"
"mere"
"Ana"
"are"
"mere"
"pere"
"oare"
"și"
"și"
"pere"
"mere"

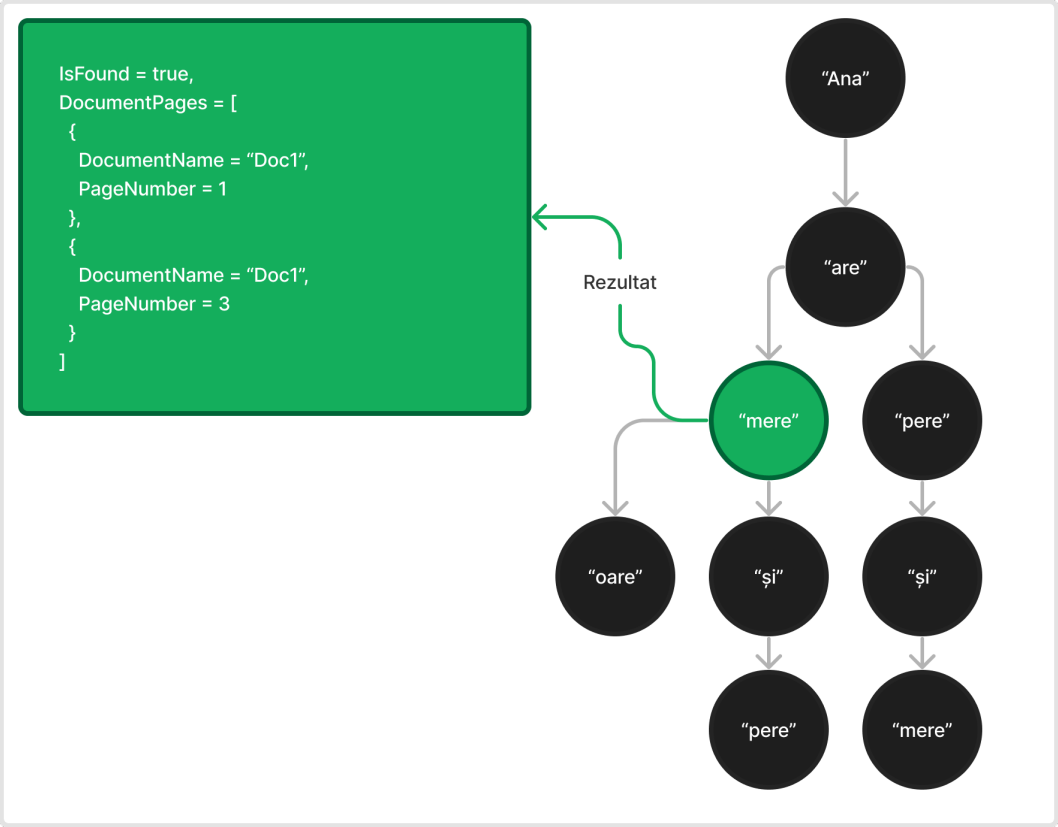


Fig. n - Exemplu de căutare a unei fraze și parcurgere a Trie-ului

5.4 Complexitatea algoritmului

Bibliografie

[1] Jay Kang, *Webpage Size - Why is it important? And how do you optimize it?*, accesat la 20.01.2024, URL: https://www.seoptimer.com/blog/webpage-size/.

[2] *Page Weight*, accesat la 20.01.2024, URL: https://httparchive.org/reports/page-weight?start=2014_03_15&end=latest&view=list.

[3] *String-searching algorithm*, accesat la 22.01.2024, URL: https://en.wikipedia.org/wiki/String-searching_algorithm.

[4] *Regular expression*, accesat la 02.02.2024, URL: https://en.wikipedia.org/wiki/Regular_expression.

[5] *Easiest way to remember Regular Expressions (Regex)*, accesat la 02.02.2024, URL: https://towardsdatascience.com/easiest-way-to-remember-regular-expressions-regex-178ba518bebd.

[6] *Finite Automata algorithm for Pattern Searching*, accesat la 25.01.2024, URL: https://www.geeksforgeeks.org/finite-automata-algorithm-for-pattern-searching/?ref=lbp.

[7] *Boyer–Moore string-search algorithm*, accesat la 05.02.2024, URL: https://en.wikipedia.org/wiki/Boyer%E2%80%93Moore_string-search_algorithm.

[8] *Boyer Moore Algorithm for Pattern Searching*, accesat la 05.02.2024, URL: https://www.geeksforgeeks.org/boyer-moore-algorithm-for-pattern-searching/.

[9] *Boyer Moore Algorithm — Good suffix heuristic*, accesat la 10.02.2024, URL: https://www.geeksforgeeks.org/boyer-moore-algorithm-good-suffix-heuristic/.

[10] Mehul Pandey, *Z Algorithm*, accesat la 11.02.2024, URL: https://www.scaler.com/topics/data-structures/z-algorithm/.

[11] *Z algorithm (Linear time pattern searching Algorithm)*, accesat la 11.02.2024, URL: https://www.geeksforgeeks.org/z-algorithm-linear-time-pattern-searching-algorithm/.

[12] *KMP Algorithm for Pattern Searching*, accesat la 14.02.2024, URL: https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/.

[13] Girish Budhwani, *KMP Algorithm (String Matching) Demystified*, accesat la 14.02.2024, URL: https://binary-baba.medium.com/string-matching-kmp-algorithm-27c182efa387.

[14] Karleigh Moore și Alex Chumbley și Jimin Khim, *Rabin-Karp Algorithm*, accesat la 23.02.2024, URL: https://brilliant.org/wiki/rabin-karp-algorithm/.

[15] *Rabin-Karp Algorithm for Pattern Searching*, accesat la 23.02.2024, URL: https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching/.

[16] Vikash Singh, *Replace or Retrieve Keywords In Documents at Scale*, accesat la 28.02.2024, URL: https://arxiv.org/abs/1711.00046.

[17] Padrig Jones, *34 Eye-Opening Google Search Statistics for 2024*, accesat la 28.02.2024, URL: https://www.semrush.com/blog/google-search-statistics/.

[18] *What is OCR (Optical Character Recognition)?*, accesat la 01.04.2024, URL: https://aws.amazon.com/what-is/ocr/.

[19] *TensorFlow*, accesat la 01.04.2024, URL: https://en.wikipedia.org/wiki/TensorFlow.

[20] *Forma unui Tensor*, accesat la 01.04.2024, URL: https://editor.analyticsvidhya.com/uploads/18560scalar-vector-matrix-tensor.jpg.

[21] *keras-ocr*, accesat la 01.04.2024, URL: https://pypi.org/project/keras-ocr/.

[22] *PyTorch*, accesat la 01.04.2024, URL: https://en.wikipedia.org/wiki/PyTorch.

[23] JaidedAI, *EasyOCR*, accesat la 01.04.2024, URL: https://github.com/JaidedAI/EasyOCR.

[24] *BenchmarkDotNet*, accesat la 03.03.2024, URL: https://benchmarkdotnet.org/.

[25] Ben Lutkevich, *Google algorithms explained: Everything you need to know*, accesat la 20.03.2024, URL: https://benchmarkdotnet.org/.

[26] Google Search Central, *How Google Search indexes pages*, accesat la 20.03.2024, URL: https://www.youtube.com/watch?v=pe-NSvBTg2o.

[27] *Search engine optimization*, accesat la 20.03.2024, URL: https://www.optimizely.com/optimization-glossary/search-engine-optimization/.

[28] *Singleton*, accesat la 30.03.2024, URL: https://refactoring.guru/design-patterns/singleton.

[29] *Associative array*, accesat la 30.03.2024, URL: https://en.wikipedia.org/wiki/Associative_array.

[30] *Trie Data Structure - Explained with Examples*, accesat la 30.03.2024, URL: https://www.studytonight.com/advanced-data-structures/trie-data-structure-explained-with-examples.

[31] Erik Demaine și Jason Ku și Justin Solomon, *Lecture 10: Depth-First Search*, accesat la 30.03.2024, URL: https://ocw.mit.edu/courses/6-006-introduction-to-algorithms-spring-2020/f3e349e0eb3288592289d2c81e0c4f4d_MIT6_006S20_lec10.pdf.

[32] Prateek Garg, *Basics of Hash Tables*, accesat la 01.04.2024, URL: https://www.hackerearth.com/practice/data-structures/hash-tables/basics-of-hash-tables/tutorial/.