UNIVERSITATEA DIN BUCUREŞTI
FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ
DEPARTAMENTUL DE INFORMATICĂ
SPECIALIZAREA SECURITATE ŞI LOGICĂ APLICATĂ

Extragerea și sumarizarea informației din pagini web

Coordonator ştiinţific
Asist.dr. Mihăiță Drăgan

Absolvent
Nicolae Cristian-Cătălin

BUCUREŞTI, Iunie 2020

Rezumat

Extragerea de informații din pagini web este extrem de importantă într-o lume în care acest sistem este în creștere continuă. Există două metode principale de extragere a informației din pagini web, metoda statică și metoda dinamică. Extragerea statică implică mai multe date de intrare decât un simplu URL deoarece utilizatorul trebuie să inspecteze conținutul HTML al paginii și să observe modele care se repetă în șabloanele cu care sunt construite paginile. Extragerea statică este foarte vulnerabilă la schimbări în șablonul paginii. Extragerea dinamică a informației este un subiect foarte larg ce poate fi abordat din mai multe unghiuri: metode bazate de pe natura arborescentă a HTML-ului, tehnici de procesare a limbajului natural, algoritmi de vedere pe calculator sau învățare automată. Pentru majoritatea paginilor problema se împarte în doua părți: găsirea șablonului paginilor din care vrem să extragem conținut și eliminarea textului irelevant precum reclame, textul de pe elementele de interfață din pagini sau codul JavaScript din pagină.

După extragerea de conținut, sumarizarea acestuia este ideală pentru a putea vedea articole diferite cu informația esențială extrasă din fiecare. La fel ca și extragerea de informații, sumarizarirea conţinului poate fi îndeplinită prin două metode principale: metoda abstractiva (bazată pe generarea de text cu metode de învățare automada) și extractivă (alegerea unei submulțimi din conținutul original). Sumarizarea extractivă este bazată pe tehnici de procesare a limbajului natural ce pot împărți textul de intrare în propoziții și mai departe în cuvinte, oferind posibilitatea de a compara propozițiile și a avea un indice de similaritate. Apoi, o metodă bună de a selecta cele mai relevante propoziții este algoritmul TextRank, algoritm bazat pe PageRank, metoda faimoasă ce stă la baza companiei Google. Conținutul de ieșire după etapa de extragere și cea de sumarizare poate fi pus înapoi în limbaj HTML pentru a oferi o versiune clară, scurtă și curată (fară reclame) a paginilor originale.

Cuprins

Introducere

Dezvoltarea mediului web încă are o creștere exponențială și cantitatea de informație conținută în acest mediu este incredibilă. Oamenii nu pot parcurge toată informația de care au nevoie, din toate sursele pe care le folosec, în timp util. Metoda prin care aceștia pot parcurge mai multă informație într-un timp mai scurt este extragerea de informații. Există numeroase pagini web ce agregă informații de pe mai multe alte pagini ce comercializează produse, oferind variante de prețuri diferite pentru ca oamenii să facă o alegere în cunoștință de cauză. La un nivel mult mai larg, chiar și Google parcurge și indexează toate paginile web pe care le întâlnește, colectând cuvinte cheie pentru a le putea potrivi cu căutările utilizatorilor.

Metodele de extragere a informației sunt diverse, dar intră în două mari categorii: extragere statică și dinamică. Extragerea se bazează pe observarea modelului în care este aranjată informația în pagina web țintă. De exemplu, putem observa că toată informația relevantă este ținută în etichete HTML de tip <DIV> ce au un anumit nume. Astfel, putem descărca conținutul HTML al paginii web și putem extrage doar textul ce se află între etichetele interesante. Acest caz este mult prea simplu și va fi întâlnit în practică, dar până și acesta este foarte vulnerabil la schimbări. Dacă dezvoltatorii paginii web trec la o altă versiune și schimbă numele etichetelor, algoritmul de extragere statică nu va mai returna informație. De cele mai multe ori, fiecare element va avea un nume diferit și informația va fi plasată în etichete diferite (<DIV>, <p>, <h1>). Astfel, chiar dacă vom identifica un model, va fi nevoie să scriem cod pentru a putea extrage informația. Iar acest cod va fi specific unui anumit șablon de pagini de pe un anumit site și nu va putea generaliza deloc pe alte siteuri.

Extragerea dinamică implică mult mai mult efort inițial, dar acesta poate generaliza și în majoriatea cazurilor va fi rezistentă la schimbări în structura HTML a șablonului. Extragerea dinamică poate fi abordată din mai multe unghiuri și include multe ramuri ale informaticii. Primul pas este obținerea Document Object Model (DOM) printr-o cerere Hypertext Transfer Protocol (HTTP). În toate browserele web existente, când un document este încărcat, DOM-ul este creat. Apoi, structura arborescentă de tip HTML poate fi parcursă pentru extragerea informației. Fac excepție de la această regulă câteva tehnologii moderne precum ReactJS ce construiesc paginile web prin programe JavaScript, oferind doar un nod HTML la momentul cererii, împreuna cu programele ce vor construi pagina în partea clientului (spre deosebire de abordarea clasică în care serverul construiește pagina și o

oferă clientului). Pentru obținerea continului HTML și construirea paginii sumarizate vom folosi librăria AngleSharp[1] scrisă în C#.

Primele studii asupra extragerii dinamice au plecat de la structura de arbore a DOM-ului[2][3] folosind structuri de date și algoritmi specifici precum compararea a doi arbori de sus în jos și încercarea a obține o mapare unu la unu între nodurile acestora. Această abordare a obținut rezultate, însă se încadrează în clasa metodelor foarte rigide, în special în prezent, când complexitatea șabloanelor este foarte ridicată și putem foarte ușor să avem noduri ce strică maparea directă dintre două pagini ce aparțin aceluiași șablon.

Expresiile regulate au fost de asemenea studiate și folosite pentru sarcina de extragere dinamică. Acestea sunt perfecte pentru a oferi un set de reguli clare pentru conținutul paginilor. Totuși, pentru a evita problemele extragerii statice de generalizare inadecvată, este nevoie de un set de reguli aproape imposibil de atins.

O abordare foarte diferită, ce evită procesarea directă a DOMului în formă de text este bazată de algoritimi din domeniul vederii pe calculator. Înainte de a putea folosi aceste metode, vom avea nevoie să folosim o tehnologie precum Selenium, tehnologie ce construiește pagina în modul grafic din DOMul primit inițial. Apoi, pot fi aplicați algoritmi de recunoaștere a caracterelor pentru a citi textul din pagină. Acești algoritmi vor avea o rată de succes ridicată datorită modului uniform în textul este reprezentat în majoritatea paginilor. Totuși, aceasta metodă va citi și conținut irelevant precum reclame sau comentarii de la utilizatori. În general, pentru a rezolva aceasta problemă se aplică diferite euristici precum faptul că informația se află în centrul paginii, deci marginile nu vor fi analizate. O altă euristică utilă este distanța de la partea de sus a paginii până la textul curent, putând astfel să deducem titlul și să împărțim pagina în mai multe zone.

Cele mai moderne abordări ale acestei probleme se află în domeniul învățării automate. Există abordări din domeniul învățării supervizate în care informația este extrasă din foarte multe pagini în mod manual și oferă algoritmului pentru etapa de învățare. Metode mult mai sofisticate există în sfera învățării nesupervizate, dar cu o rată de succes mult mai scăzută.

Pentru lucrarea de față, abordarea aleasă este bazată pe lucrarea „Web Template Extraction Based on Hyperlink Analysis.”[4], lucrare ce se bazează pe forma de graf a unui site și pe Uniform Resource Locators (URLurile din pagini ce trimit către alte pagini). Paginile sunt reprezentate drept noduri în graf și linkurile care trimit o pagină la alta sunt reprezentate drept muchii între noduri. Identificarea șablonului pentru paginile de tip articol este făcută printr-o euristică foarte puternică,

aceasta fiind faptul că toate aceste pagini pot ajunge înapoi pe pagina principală într-un singur pas (adică fiecare pagină cu conținut interesant conține un URL către pagina principală), URL prezent în meniul din partea de sus a paginii. Abordarea noastră diferă totuși în mai multe privințe, încercând să treacă de unele limitări prezente în lucrarea inițială.

Extragerea informației este totuși doar primul pas deoarece, cum am menționat deja la unele abordări, prima mulțime de informații extrasă de oricare dintre acești algoritmi va conține mult text irelevant precum reclame, cod JavaScript folosit pentru comportamentul dinamic al paginilor sau textul ce apare pe elementele de interfață din pagină. Definirea informației relevante este dificilă, fiind mai ușor să alegem euristici ce identifică informația irelevantă analizând informația din pagină la un nivel discretizat (de exemplu, la nivel de nod HTML).

Pentru aceasta etapă vom pleca de la abordarea prezentată în lucrarea „Main Content Extraction from Web Pages Based on Node Characteristics”[5], lucrare ce prezintă doi factori ce pot fi calculați la nivel de nod HTML, densitatea textului și densitatea de URLuri, factori ce pot fi folosi pentru a tria mulțimea de noduri și a le selecta doar pe cele ce trec de un anumit prag. Acești doi factori au dat rezultate bune, deși alegerea valorilor de prag pentru selecția nodurilor s-a dovedit foarte rigidă și a trebuit să fie lăsată mai permisivă deoarece cu valori inadecvate algoritmul elimină mult conținut relevant. Pentru a ameliora problemele cauzate de rigiditatea valorilor de prag din acest algoritm, acestea au fost setate cât mai permisiv, astfel încât întregul conținut relevant să fie păstrat, urmând ca apoi să prelucrăm cu tehnici din sfera procesării limbajului natural.

Partea de sumarizare din lucrare va avea ca bază algoritmul TextRank, prezentat în lucrarea „Bringing Order Into Texts” de Rada Mihalcea și Paul Tarau [7]. Lucrarea pornește de la celebrul algoritmul PageRank[6], algoritm creat de fondatorii Google, Sergey Brin și Larry Page și folosit pentru a ordona paginile web după un anumit rang, bazându-se din nou pe structura lor de graf. Algoritmul TextRank extinde PageRank la propoziții, oferind propozițiile cele mai relevante dintr-o mulțime de propoziții date ca intrare sub formă de graf ponderat, unde ponderile sunt indici de similaritate între propoziții. Acest coeficient de similaritate va fi calculat cu ajutorul modelului de învățare automată Word2Vec, model ce mapează cuvinte în spații N-dimensionale, unde fiecare cuvânt are asignată o poziție în spațiu. Putem aplica operații matematice precum scăderea pozițiilor cuvintelor pentru a obține vectori și putem refolosi acești vectori pentru alte puncte(de exemplu: vectorul rezultat din „rege” – „regină” va putea fi adunat la valoarea punctului „bărbat” pentru a rezulta poziția

cuvântului „femeie”). După ce coeficienții sunt calculați, vom apela un proces clasic de tip PageRank pe graful de propoziții și coeficienți.

Capitolul 1 prezintă în detaliu toate noțiunile teoretice folosite de-a lungul lucrării: structura siteurilor și paginilor web, structura HTML, abordarea folosită în [4] pentru extragerea șablonului (și schimbările aduse la această abordare), metodele de filtrare a textului irelevant din [5] (și optimizările aduse pentru a nu filtra mult prea restrictiv), tehnicile de procesare a limbajului natural folosite în lucrare, algoritmul PageRank precum și metoda Word2Vec dezvoltată de cei de la Google.

Capitolul 2 va pune în evidență tehnologiile folosite în realizarea lucrării: limbajul de programare C#, librăria AngleSharp, folosită pentru obținerea conținutului HTML, librăria OpenNLP, folosită extensiv pentru separarea textului în propoziții și apoi în cuvinte, chiar și etichetarea cuvintelor ca anumite părți de vorbire. De asemenea, vom prezenta librăria ce ne permite să citim baza de date Word2Vec cu 100 de miliarde cuvinte oferită de Google și librăria ce implementează algoritmul PageRank.

Capitolul 3 pune în valoare în detaliile de implementare ale aplicației prezentând capabilitatile dezvoltate și prezente în aplicație, modul în care acestea au fost implementate, cele mai interesante probleme tehnice apărute pe parcursul dezvoltării și soluțiile create pentru a trece de acestea.

La final, vom avea o rubrică de concluzii unde vom analiza o serie metode de testare dezvoltate special pentru aplicația de față. Cu acestea, vom identifica limitările prezente și vom teoretiza posibile soluții pentru viitor.

CAPITOLUL 1 – Concepte teoretice folosite în dezvoltarea aplicaţiei

1.1 Istoricul world wide web(WWW)[9]

Tim Berners-Lee este recunoscut internațional drept creatorul tehnologiilor ce stau la baza webului într-un document publicat în Martie 1989 numit “Information Management: A Proposal”. Acesta lucra ca inginer software la acceleratorul de particule de la Geneva (CERN) și a observat greutatea cu care oamenii de știință puteau să pună la comun informațiile pe care le aveau. Internetul deja conecta milioane de calculatoare din intreaga lume, dar Tim Berners-Lee avea infrastructura ce îi permitea să pună la punct un sistem de partajarea a documentelor în întreaga lume.

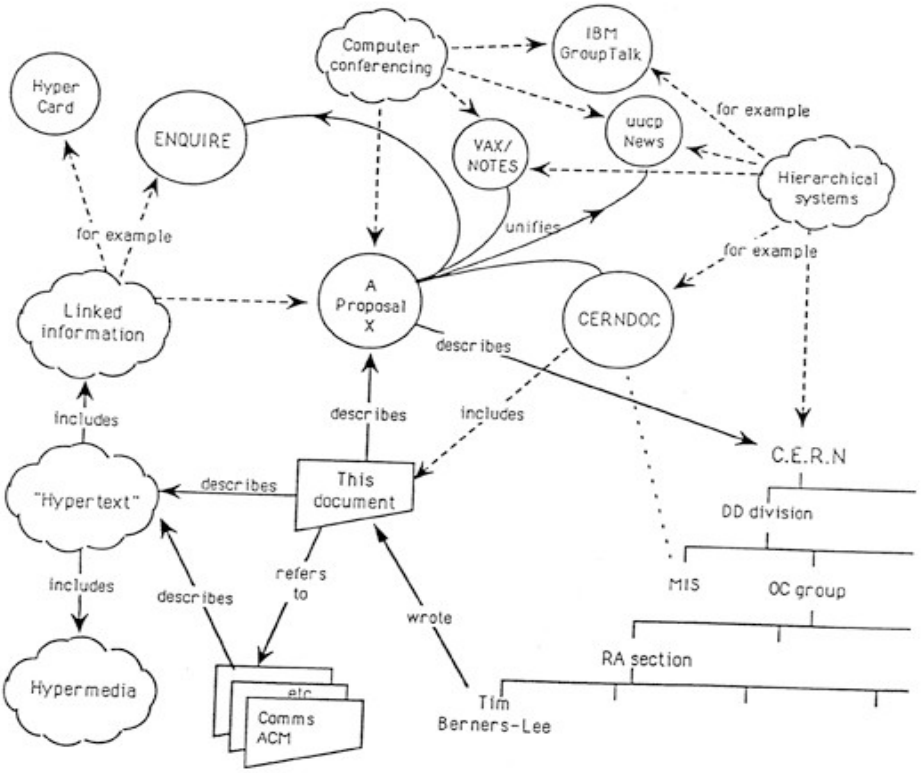

Figura 1.1. Schiţa din documentul iniţial al lui Tim Berners-Lee, ce prezintă un sistem foarte apropiat de implementarea modernă a WWW. [1]

Deși inițial ideea sa a fost refuzată, în octombrie 1990 acesta a extins documentul inițial și a definit în detaliu cele trei tehnologii necesare pentru funcționarea WWWului. Acestea sunt:

- HTML – limbajul ce defineşte structura paginilor.

- Uniform Resource Identifier(URI) – o adresă ce identifică unic o resursă din web. Termenul de URL este de asemenea folosit cu acelaşi scop.
- Hypertext Transfer Protocol(HTTP) – protocol ce permite regăsirea resurselor din web.

De asemenea, Tim Berners-Lee a scris și primul browser (“WorldWideWeb.app”) și primul server web (httpd). Pentru că tehnologia dezvoltată de acesta să fie folosită de întreaga lume în mod gratuit, Tim Berners-Lee a insistat ca cei de la CERN să facă codul public, fară nici un fel de pretenții monetare asupra acestuia. Această decizie a fost luată în 1993 și rezultatele pot fi simțite în ziua de astăzi când aproximativ două din cinci persoane folosesc sistemul conceput de Tim Berners-Lee.

Principiile puse la punct de comunitatea formată în jurul sistemului web la începuturile sale au fost revoluţionare şi încă sunt protejate. Acestea sunt:

- Decentralizare: nu există o autoritate principală ce autorizează ce există pe web. Nu există un mod de a opri sistemul pentru toată lumea.
- Non-discriminare: un utilizator ce plătește pentru o conexiune la internet cu un anumit grad de calitate a serviciului va comunica în mod egal cu alt utilizator ce plătește mai mult pentru o calitate a serviciului superioară.
- Universalitate: pentru ca oricine să poată publica orice pe internet, toate calculatoarele trebuie să vorbească același limbaj, indiferent de hardwareul folosit, locația utilizatorului sau credințele culturale și politice.
- Consens: pentru a permite existența unor standarde universale, toată lumea trebuia să fie de acord asupra acestora. Tim Berners-Lee și cei din comunitate au facilitat acest lucru prin procese foarte transparente de decizie.

1.2 Structura HTML[10]

HTML este limbajul ce permite exprimarea de documente ce pot fi prezentate de către browserele web. Acestea primesc documente HTML de la servere web și le prezintă utilizatorului. HTML descrie structura semantică a unei pagini web, fiind ajutat pe parte de prezentare de Cascading Style Sheets (CSS), iar pe partea de interacțiune de limbajul JavaScript, care este un limbaj de tip client, în sensul în care acesta este livrat odată cu documentului HTML și se execută în browserul utilizatorului după încărcarea documentului. Limbajul PHP este un limbaj de tip server, deci codul PHP se va executa înainte de încărcarea conţinului HTML. Protocolul prin care documentele HTML sunt transmise este HTTP, deși acesta poate trimite și imagini, sunet sau alt fel de conținut.

Un document HTML are o structura arborescentă, începând de sus cu un singur nod ce are mai multe noduri copil. Frunzele acestui arbore sunt elemente HTML de bază, precum controale grafice, paragrafe de text sau imagini. Elementele HTML vin de obicei în perechi de etichete de tipul “<p>” şi “<./p>”, unde prima etichetă denotă începutul elementului și a doua finalul său. Între aceste două etichete există atributele elementului, care subliniază anumite proprietăți pe care elementul le va avea când este prezentat în browser. Există și elemente HTML care nu sunt descrise între perechi de etichete precum “”. Un exemplu de document minimal este următorul:

```
<!DOCTYPE html>
<html>
        <head>
            <title>Titlu</title>
        </head>
        <body>
            <p>Conţinut text</p>
        </body>
</html>
```

Prima linie arată faptul că documentul va fi tratat dupa standardul HTML5. Conţinutul dintre etichetele “<html>” şi “</html>” descrie pagina web, iar conţinul dintre etichetele “<body>” şi “</body>” descrie partea vizibilă din pagina, pe când eticheta “<head>” este folosită pentru a alege titlul paginii.

Nu vom intra în toate detaliile de sintaxă ale limbajului, deoarece nu toate sunt relevante pentru extragerea de informații, dar o problemă întâmpinată în timpul extragerii conținutului de text irelevant este faptul că unele noduri pot conține cod JavaScript, care, cum a fost menționat deja, este folosit

pentru a face pagina web să fie interactivă. Abordarea pe care o vom folosi pentru extragerea informației din noduri prelucrează fiecare nod și calculează anumite euristici după care ia decizii de triere. Astfel, unele noduri ce conțin doar cod JavaScript drept text pot fi fals pozitive (noduri ce doar par să conțină text relevant).

```
<script>
                document.getElementById("header").innerHTML = "Titlu nou";
</script>
```

În exemplul de mai sus putem vedea eticheta “<script>” în care se include de obicei cod JavaScript, cod care în cazul de față caută după nume nodul “header” pentru a îi schimba conținutul.

În prezent există foarte multe sisteme complexe de gestiune a conținutului HTML pentru a optimiza dimensiunea paginilor web și cantitatea de informație ce trebuie transmisă. Un astfel de sistem este ReactJS, dezvoltat de Facebook. Acesta construiește conținutul HTML din cod JavaScript și livrează doar un singur nod HTML și multe script-uri JavaScript către client, unde acestea vor fi rulate pentru a se construi pagina. Un exemplu:

```
const element = <h1>Titlu</h1>;
ReactDOM.render(element, document.getElementById('root'));
```

În acest exemplu se adaugă dinamic o etichetă de tip “<h1>” în singurul nod HTML prezent la începutul nodului rădăcină. Această abordare este foarte dinamică și eficientă, dar pune probleme extragerii de informație deoarece pentru a avea informația, trebuie să rulăm codul Javascript primit, la fel ca un browser obișnuit.

1.3 Colectarea de pagini similare dintr-un website [4]

Lucrarea [4] propune o metodă foarte simplă și puternică pentru colectarea unei mulțimi de pagini dintr-un website, mulțime din care putem încerca să extragem un șablon. Putem reprezenta un website ca un graf, unde fiecare nod este o pagină și fiecare muchie este un URL ce duce către o altă pagină din website. Această reprezentare ne ajută deoarece putem avea o vedere de ansamblu asupra topologiei siteului și putem aplica diverse observații și euristici asupra acesteia.

O presupunere pe care o putem face este faptul că toate paginile de conținut din website au un meniu în partea de sus a paginii, meniu care este obicei plin cu URLul ce duc către alte pagini cheie din site printre care și pagina de start. Astfel, dacă dăm algoritmului o pagină de start, care de obicei va fi pagina principală, atunci putem construi mulțimea de pagini de conținut din paginile pe care am ajuns de la pagina principală și care se pot întoarce înapoi pe aceasta printr-un URL. Aceasta mulțime va fi un subgraf în interiorul topologiei site-ului.

În figura 1.2 putem vedea un astfel de exemplu. Nodurile negre reprezintă paginile spre care arată URL-urile din meniu. Pentru că meniul este omniprezent, putem ajunge din orice pagină spre oricare alta pagină din meniu, deci se formează un graf complet. Același lucru se aplică și pentru paginile la care putem ajunge cu URL-urile din alte părți ale meniului, reprezentate în figura prin nodurile gri. Apoi avem nodurile albe, care sunt pagini la care putem ajunge urmând URL-uri din paginile de meniu.

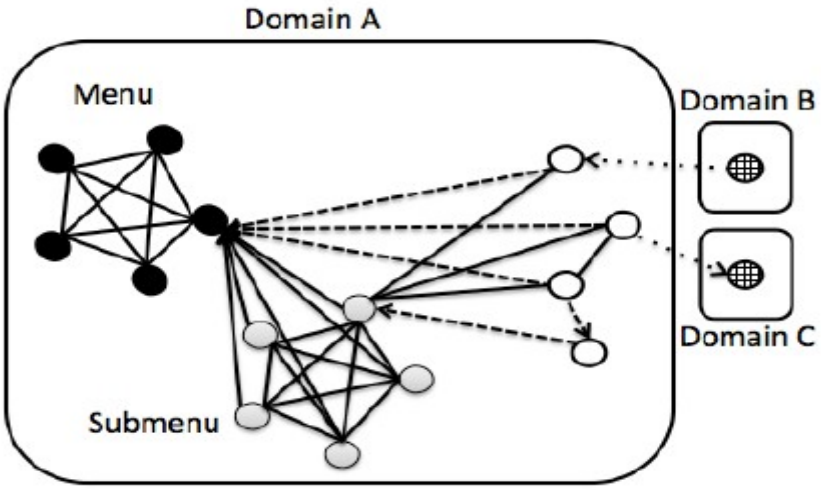

Figura 1.2. Topologia unui site obişnuit. [2]

Se poate deci observa că putem construi un algoritm ce găsește o mulțime de pagini, adică un subgraf din graful tuturor paginilor din website. Algoritmul propus de autori este urmatorul:

Algorithm 1 Extract a n-CS from a website

Input: An *initialLink* that points to a webpage and the expected size n of the CS.
Output: A set of links to webpages that together form a n-CS.
If a n-CS cannot be formed, then they form the biggest m-CS with $m < n$.

begin
$keyPage = loadWebPage(initialLink)$;
$reachableLinks = getLinks(keyPage)$;
$processedLinks = \emptyset$;
$connections = \emptyset$;
$bestCS = \emptyset$;
foreach *link* **in** *reachableLinks*
$webPage = loadWebPage(link)$;
$existingLinks = getLinks(webPage) \cap reachableLinks$;
$processedLinks = processedLinks \cup \{link\}$;
$connections = connections \cup \{(link \rightarrow existingLink) \mid existingLink \in existingLinks\}$;
$CS = \{ls \in \mathscr{P}(processedLinks) \mid link \in ls \wedge \forall l, l' \in ls \,.\, (l \rightarrow l'), (l' \rightarrow l) \in connections\}$;
$maximalCS = cs \in CS$ such that $\forall cs' \in CS \,.\, |cs| \geq |cs'|$;
if $|maximalCS| = n$ **then return** *maximalCS*;
if $|maximalCS| > |bestCS|$ **then** $bestCS = maximalCS$;
return *bestCS*;
end

Figura 1.3. Algoritmul de extragere a unui subgraf de mărime N dintr-un website. [3]

Având un URL inițial ca dată de intrare, se obține DOMul pentru ca apoi să se extragă toate URLurile prezente în acesta. Apoi se itereaza prin toate acest URLuri și pentru fiecare, se încarcă pagina și toate URL-urile de pe pagina respectivă. Făcând intersecția dintre URLurile prezente pe pagina inițială și cele prezente pe pagina de la iterația curentă, vom găsi toate legăturile dintre aceste pagini, deci pentru fiecare element din intersecție, vom defini o conexiune (ce are un nod start și un nod sfârșit). La fiecare pas, pe baza conexiunilor, putem avea o funcție ce returnează toate subgrafurile ce sunt formate din acele conexiuni. Când găsim un subgraf de dimensiune N sau unul de dimensiune mai mare decât N, algoritmul se poate opri.

Acest algoritm dă rezultate bune, cum este prezentat în următorul capitol când vom discuta implementarea acestuia. Două probleme pe care le prezintă totuși pot fi:

- Metoda de căutare a subgrafurilor este destul de complexă din punct de vedere computațional.
- Se testează foarte multe URLuri, deci în practică apar și probleme din cauza încărcării rețelei.

- Pe siteuri cu foarte multe UR-uri prezente în fiecare pagină, în special pe pagina data ca intrare, algoritmul poate să fie deviat de numărul mare de subgrafuri corespunzătoare și să returneze un subgraf care nu conține doar pagini cu conținut relevant. De exemplu, pagini auxiliare precum pagina de contact a siteului pot fi luate în același subgraf cu pagini cu conținut relevant, îngreunând procesul de obținere a șablonului.

Chiar dacă aceste limitări există, algoritmul (cu câteva optimizări pe care le vom prezenta la partea de implementare) este cel folosit pentru rezolvarea problemei de găsire a unei mulțimi de pagini înrudite din care să încercăm extragerea unui șablon.

1.4 Extragerea şablonului folosit pentru construirea paginilor dintr-un site web[4]

Lucrarea [4] prezintă un mod de extragere a şablonului dintr-o pagină web, şablon folosit apoi pentru a identifica paginile ce conţin informaţie relevantă. Autorii estimează că 40-50% din toată informaţia de pe web este reprezentată de şabloane.

Abordarea se bazează pe informaţia din Document Object Model (DOM). DOMul este o mulţime de obiecte folosite pentru reprezentarea documentelor HTML şi Extensible Markup Language (XML). DOM-l are o structură de arbore, cum am prezentat anterior.

Autorii definesc arborele ca $T=(N,E)$ unde N este o mulţime finită de noduri (cele din DOM) şi E o mulţime finită de muchii (legăturile părinte-copil dintre noduri). Apoi se defineşte noţiunea de "mapare" între doi arbori, adică o corespondenţă la nivel de nod.

O mapare între aborele $T=(N,E)$ şi arborele $T'=(N',E')$ este o mulţime M de perechi de noduri de tipul $(n,n'),n\in N,n'\in N'.$ Se mai defineşte şi ca $root(T)$ care este rădăcina arborelui T şi $parent(n)$ care este părintele nodului n .

Astfel, pentru orice două perechi avem $(n_1,n'_1),(n_2,n'_2)\in M,n_1=n_2 \; iff \; n'_1=n'_2.$

Definiţia mapării permite introducerea conceptului de mapare sus-jos. Dată o relaţie de egalitate între două noduri din arbore, o mapare M dintre doi arbori T şi T' este egală sus-jos dacă şi numai dacă:

- pentru orice pereche $(n,n')\in M,nRn'.$
- pentru orice pereche $(n,n')\in M$ cu $n\neq root(T)$ şi $n'\neq root(T')$ există de asemenea o pereche $(parent(n),parent(n'))\in M.$

Practic, fiecare nod dintr-un arbore trebuie să aibă un corespondent direct în celălalt arbore, iar relația de corespondență este liberă pentru a fi implementată în moduri diferite. Un mod intuitiv și direct ar fi să considerăm două noduri echivalente dacă acestea sunt același tip de element HTML (paragraf, link). Autorii folosesc o relație de corespondentă mai complexă bazată de atribute din fiecare nod HTML.

Putem vedea în figura 1.2 un exemplu de mapare între doi arbori echivalenți. Încă de la rădăcină, unde avem nodul "body" și până la frunzele arborelui, adică cele două paragrafe și un tabel, avem o corespondență de tip de element HTML. Alte condiții mai complexe pot cere de exemplu ca elementele să aibă același atribut "classname".

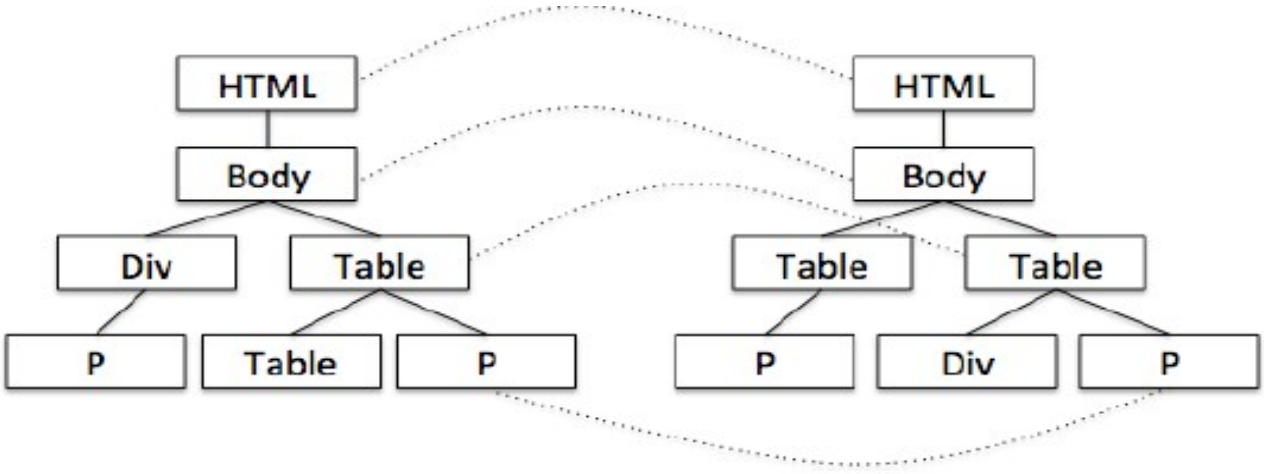

Figura 1.4. Un exemplu de mapare sus-jos între doi arbori. [4]

Într-un final, avem descrierea formală a unui şablon pentru pagini web. Dacă avem p_0 o pagină web cu arborele DOM asociat T_0 şi o mulţime de pagini $P=\{p_1 \dots p_n\}$ cu arborii asociaţi $\{T_1 \dots T_n\}$. Un şablon al lui p_0 faţă de P este un arbore (N,E) unde:

- nodurile $N=\{n \in N_0 | \forall 1 \le i \le n, (n,\_) \in M_{(T_0,T_i)}\}$ unde $M_{(T_0,T_i)}$ este o mapare sus-jos între arborii T_0 si T_1 .
- Muchiile $E=\{(m,m') \in E_0 | m,m' \in N\}$.

Astfel, formalizarea șablonului este o nouă pagină care este egală sus-jos cu toate paginile dintr-o anumită mulțime (pagini din același website).

Abordarea prezentată până acum din lucrarea [4] oferă rezultate bune, dar după implementarea acesteia, ea s-a dovedit destul de rigidă, chiar și cu cea mai puțin restrictivă relație de corespondență

între două noduri pe care o putem alege, cea în care nodurile trebuie doar să aibă același tip. Rigiditatea stă în faptul că șabloanele moderne sunt suficient de dinamice încât să găsim frunze lipsă într-o instanță a șablonului față de altă instanță a sa, iar acest factor nu ar trebui să elimine complet posibilitatea ca cele două instanțe să fie înrudite.

Lucrarea propune o altă metodă prin care putem extrage un șablon dintr-o mulțime de arbori DOM. Începând de la totalitatea subgrafurilor obținute prin metoda de la subcapitolul anterior (toate subgrafurile de dimensiune N sau mai mare), vom aplica diverse metode pentru a obține subgraful care conține cele mai similare pagini, ca structura DOM, deci subgraful care este cel mai probabil să fie compus din pagini care au la bază același șablon.

Pentru fiecare pagină din mulțime, vom construi un vector de frecvență care va număra de câte ori apare o anumită etichetă HTML în aceasta. De exemplu, pentru documentul HTML de mai jos, avem perechile eticheta-frecvență: (<head>, 1), (<body>, 1), (<p>, 2) şi (<a>, 1).

```
<!DOCTYPE html>
<html>
        <head>
            <title>Titlu</title>
        </head>
        <body>
            <p>Conţinut text</p>
            <p>Conţinut text</p>
            <a>URL</a>
        </body>
</html>
```

Documentele HTML complexe vor avea zeci de etichete, iar cele mai frecvente etichete din pagini orientate pe text vor apărea de sute de ori (precum eticheta <p>).

Abordarea de reprezentare a DOMului drept vector de frecvență, față de abordarea autorilor din lucrarea [4], în forma naturală a DOMului de arbore, este motivată de dorința a reduce rigiditatea căutării șablonului. Dacă avem 4 pagini ce sunt instanțe ale aceluiași șablon, este foarte probabil ca aceastea să difere parțial, în special în partea de jos a arborelui, cu cât ne apropiem de frunze (care în cazul HTML ar însemna că o pagină conține mai mult text decât altă pagină, deci are mai multe etichete de tip <p>, etichete ce vor fi mereu frunze). De asemenea, o soluție care ar permite ajustarea rigidității cu care facem căutarea ar fi ideală pentru a putea rula programul în mai multe moduri.

Astfel, dintr-o mulțime cu N pagini, deci N DOMuri, vom obține N vectori de frecvență. Pentru a duce vectorii la aceeași lungime (este posibil ca unele pagini să aibă elemente HTML în plus sau în minus față de celelalte) vom introduce elementele lipsă în fiecare vector, cu valoarea de frecvență zero.

Cum am menționat, dorim să reducem totul la o singură valoare ce definește cât de similare sunt paginile din mulțime. O metodă este să calculăm deviația standard pentru cele N valori de frecvență ale fiecărui element HTML. Apoi, media valorilor de deviație standard pentru fiecare element HTML va reprezenta valoarea finală ce definește similaritatea dintre paginile din mulțime. O valoare cât mai mică va sugera o mulțime de pagini similare. Dacă valoarea aceasta este sub un prag ales, atunci șablonul va fi alcătuit din mediana valorilor pentru fiecare element HTML, deoarece media aritmetică ar fi foarte afectată de valori diferite de restul (“outlier”).

De exemplu, dacă avem următorii 4 vectori de frecvență (fără valorile ce sunt constante și prezente în orice document: (<head>, 1), (<body>, 1)):

1. [(<p>, 5), (<a>, 2), (<s>, 2), (
, 2), (<script>, 1)].
2. [(<p>, 4), (<a>, 2), (<s>, 1), (
, 2), (<script>, 0)].
3. [(<p>, 4), (<a>, 3), (<s>, 2), (
, 2), (<script>, 0)].
4. [(<p>, 5), (<a>, 2), (<s>, 2), (
, 2), (<script>, 1)].

Eticheta <script> apare doar în 2 documente, deci în celealte 2 o vom introduce noi, dar cu valoarea zero. Vectorii de frecvenţă şi deviaţiile standard pentru fiecare element HTML în cele 4 documente sunt:

- <p> - [5, 4, 4, 5] – deviaţia standard 0.57.
- <a> - [2, 2, 3, 2] – deviaţia standard 0.50.
- <s> - [2, 1, 2, 2] – deviaţia standard 0.50.
-
 - [2, 2, 2, 2] – deviaţia standard 0.00.
- <script> - [1, 0, 0 ,1] – deviaţia standard 0.57.

Media aritmetică a deviaţiilor standard este 0.428. Dacă această valoare este sub un anumit prag ales de noi, vom considera că cele 4 pagini sunt instanţe ale unui şablon şi vom construi şablonul în sine folosind mediana pentru fiecare element.

Astfel, vom sorta vectorii şi vom alege mediana:

- <p> - [4, 4, 5, 5] - mediana 4.5, dar vom folosi partea întreagă 4.
- <a> - [2, 2, 2, 3] - mediana 2.
- <s> - [1, 2, 2, 2] – mediana 2.

-
 - [2, 2, 2, 2] – mediana 2.
- <script> - [0, 0, 1 ,1] – mediana 0.5, dar vom folosi partea întreagă 0.

Șablonul considerat de noi va fi vectorul de frecvenţă:

[(<p>, 4), (<a>, 2), (<s>, 2), (
, 2), (<script>, 0)]

Abordarea aceasta scapă de rigiditatea metodei cu compararea sus-jos deoarece similaritatea DOMurile este dată de o singură valoare, media deviațiilor standard pentru frecvențele elementelor HTML. Când primim o nouă pagină și vrem să verificăm dacă este o instanță a șablonului, este ușor să reperăm procesul cu doar două pagini.

Dezavantajul este că se pierde structura directă de arbore, deci nu mai avem informații despre relațiile părinte-copil dintre elementele din DOM. Metoda inițială avea ca intrare o relație de comparare între noduri, dar metoda noastră introduce o valoare de prag aleasă empiric, lucru care poate să introducă destul de multă varianță, deoarece această valoare poate fi foarte diferită de la un website la altul.

1.5 Filtrarea conţinutul irelevant din pagină [5]

O mare parte din conținutul pe care îl vom întâlni în majoritatea paginilor web va fi irelevant (text prezent pe controale grafice precum butoane, cod JavaScript, reclame). Pentru a menține o calitate ridicată a conținutul extras din pagină, sunt necesare metode de filtrare a conținutului irelevant. Lucrarea [5] propune un algoritm estimativ, la nivel de nod în DOM, ce judecă dacă nodul respectiv este nod ce conține text relevant sau este doar zgomot.

Nodurile dintr-un DOM se pot împărți în mai multe categorii în funcție de scopul etichetelor acestora în pagină. Câteva categorii pot fi:

- Etichete pentru interacțiune. Acestea se ocupă cu legătura utilizatorului cu serverul sau pur și simplu schimbarea dinamică a aspectului paginii. Exemple de astfel de etichete: <script>, <applet>, <object> etc.
- Etichete pentru stil. Acestea se ocupă de schimbarea aspectului paginii şi a conţinutului. De exemplu <b> pentru text îngroşat.
- Etichete pentru metadate. Aceastea descriu elementele de bază pentru un document HTML: <head>, <title>, <meta> etc.
- Etichete de tip container. Acestea aranjează conţinutul paginii şi de cele mai multe ori, conţinutul esenţial din pagină va fi prezent în noduri din arborii ce pornesc de la acest tip de etichete. De exemplu: , <table>, , <p>, <td> etc.

Pentru că autorii vor să eficientizeze conținutul scanat pentru filtrare, algoritmul se va concentra doar pe ultimul tip de etichete (pe noduri definite de acestea).

Următorul pas din lucrare este să se definească 4 niveluri de caracteristici ale unei pagini sau ale mai multori pagini (nivel macro) sau ale unui nod (nivel micro). Acestea sunt:

- Caracteristicile de bază ale unui nod precum mărimea şirului de caractere din interiorul unui nod de tip <p> sau numărul de URLuri prezente într-un nod de tip .
- Caracteristici de bază pentru toată pagina precum densitatea textului.
- Caracteristici grafice ale paginii.
- Caracterici comune ale mai multor pagini (şabloane).

Caracteristicile la nivel de nod sunt cele mai accesibile și acestea au fost alese de autori pentru această lucrare. Astfel, aceștia introduc formula de calcul a densității textului asupra unui nod b pentru cazul în care avem mai mult de o linie (delimitată de caracterul "\n"):

$$\frac{T'(b)}{(L(b)-1)*maxLen}$$

Unde $T'(b)$ reprezintă numărul de caractere fără ultima linie. $L(b)$ este numărul de linii din nodul curent, iar $maxLen$ reprezintă numărul maxim de caractere întâlnite într-o linie din pagină.

Pentru cazul în care avem o singură linie, formula densităţii textului este:

$$\frac{T(b)}{maxLen}$$

Motivul pentru care autorii ignoră ultima linie când acest lucru este posibil este datorat faptului că ultima linie poate introduce un efect fals asupra densității, având o lungime mai scurtă.

Al doilea factor la nivel de nod este densitatea de URL-uri dintr-un nod. Formula este destul de intuitivă:

$$\frac{Ta(b)}{T(b)}$$

$Ta(b)$ este numărul de caractere prezente în nodul curent ce fac parte dintr-o etichetă de tip <a>, folosită pentru URL-uri. $T(b)$ este numărul total de caractere din nodul b.

Autorii mai introduc şi noţiunea de similaritate între noduri vecine, folosindu-se de densitatea textului (Td) şi densitatea de URL-uri (Ud):

$$diferenta(a,b)=|((Td(a)-Td(b))*A)|+|((Ud(a)-Ud(b))*B)|$$

Diferența va fi suma ponderată a diferențelor celor doi factori pentru cele doua noduri. Ponderile A și B au fost alese de autori drept 0.2 și 0.8, deoarece au observat că densitatea de URL-uri este mai relevantă decât densitatea textului.

În final, algoritmul propus este următorul:

1. Obţinerea DOMului.
2. Eliminarea etichetelor ce nu au conţinut text relevant.
3. Se unesc nodurile vecine clasificate drept similare. După unirea acestora, se recalculează cei doi factori pentru noul nod.
4. Se elimină noduri pe bază unor valori de prag pentru densitatea textului și cea de URLuri.

Valorile alese de autori ca prag pentru densitatatea textului și cea de URLuri sunt 0.5 și 0.333. Astfel, dacă un nod este compus mai mult de o treime din URLuri, acesta este considerat zgomot. De asemenea, dacă un nod trece de jumătatea ecranului cu conținutul sau text, acesta este probabil un nod relevant.

Inițial, implementarea din lucrare a folosit întregul algoritm prezentat aici, iar rezultatele au fost satisfăcătoare. După numeroase teste pagini web diverse, s-a observat că pasul 3, cel care unește nodurile vecine similare, salva unele noduri irelevante prin acest proces de unire. Se întâmplă destul de des ca un nod *A* relevant să fie unit cu un nod *B* irelevant, iar nodul *C* rezultat să treacă de valorile de prag de la pasul 4. Evident, aceste cazuri se află la granițele valorilor numerice folosite în pasul 3 și pasul 4, dar aceste cazuri erau suficient de frecvente pentru a cere îmbunătățiri. Detaliile vor fi prezentate în capitolul următor, dar în general, s-a renunțat la unirea nodurilor similare, scăpând de valorile de prag pentru *A* și *B* din formula diferenței dintre două noduri. O altă modificare adusă a fost eliminarea valorilor de prag pentru densitatea textului și densitatea URLurilor, acestea calculându-se la rulare drept media aritmetică a tuturor densităților de text, respectiv URL, din mulțimea de noduri.

1.6 Procesarea limbajului natural (NLP)

În lucrare, după ce partea de filtrare a conținutului este finalizată, vom rămâne cu informație sub forma unor șiruri de caractere. Cu ajutorul unor metode din sfera procesării limbajului natural, vom segmenta informația în propoziții și apoi în cuvinte, având și posibilitatea de a găsi pentru fiecare cuvânt chiar și partea de vorbire. Segmentarea informației în propoziții și cuvinte este necesară atât pentru adăugarea unui nou strat de filtare a textului, dar și pentru a putea aplica apoi metoda de sumarizare bazată pe Word2Vec.

Pentru partea de procesare a limbajului natural se folosește exclusiv librăria OpenNLP[11], în varianta în care a fost portata pentru limbajul C# din limbajul Java (sub numele neoficial de “SharpNLP”)[12]. Cum am menționat, primul pas este împărțirea textului în propoziții. Documentația oficială nu descrie decât modul de utilizare al librăriei, neintrând în detalii foarte mari despre implementare, dar aici putem profita de faptul că librăria are codul făcut public și putem investiga direct.

Prima metodă care se execută asupra șirului dat ca intrare (concatenarea întregului text relevant din DOM) este împărțirea sa în propoziții. Împărțirea în propoziții se face pe baza algoritmului de segmentare “Maximum Entropy Markov Model” (MEMM)[13]. Pe scurt, această metodă va împărți subșirul în diferite predicate și le va evalua probabilitatea ca acestea să fie un subșir ce are sens. Evident, modelul ce evaluează predicatele este un model antrenat pe o vocabularul unei anumite limbi, în cazul de față, limba engleză.

A doua metodă care se execută asupra fiecărei propoziții obținute la pasul anterior este împărțirea acestora în bucăți pe baza spațiilor, astfel obținem un vector de perechi (X, Y) unde X este indexul de început al subsirului și Y este indexul de final. Această separare este simplă, dar rezultatele nu vor fi foarte precise, deci fiecare astfel de pereche este apoi luată în parte. O optimizare este posibilă pentru subșirurile alfa-numerice, acestea sunt validate direct. Șirurile ce nu sunt complet alfa-numerice vor intra în algoritmul de segmentare . De exemplu, pentru subșirul “Zero-Day”, predicatele sale vor fi următoarele: "p=Zero-D", "s=ay", "p1=D", "p1_alpha", "p1_caps", "p2=-", "p21=-D", "p1f1=Da", "f1=a", "f1_alpha", "f2=y", "f2_alpha", "f12=ay". Procesul de evaluare a predicatelor implică o căutare destul de complexă, pentru că sunt evaluate toate subșirurile posibile pentru “Zero-Day”. Astfel, optimizarea legată de subșirurile alfa-numerice ajută semnificativ timpul de execuție.

Ultima metodă aplicată este cea care etichetează fiecare cuvânt din propoziție cu o parte de vorbire. Se folosește tot un model bazat pe entropie maximă antrenat pe limba engleză prin metoda descrisă în [14]. Vom folosi rolul cuvintelor ca părți de vorbire pentru a îmbunătăți filtrarea prin eliminarea de propoziții fară verbe.

1.7 Term frequency – inverse document frequency (TF-IDF[15])

Un alt concept important, folosit în lucrare pentru găsirea celor mai importante cuvinte din document este TF-IDF. Găsirea unei mulțimi de cuvinte relevante pentru un anumit document este utilă pentru a identifica documentul și tematica sa, dacă construim o bază de date sub forma unui dicționar cu cheia drept URLul paginii și valoarea fiind vectorul de cuvinte relevante. De asemenea, această

mulțime de cuvinte poate ajuta și la selectarea propozițiilor importante după ce le vom sorta cu o metodă bazată pe Word2Vec ce va fi prezentată în capitolul următor.

Frecvența unui termen (TF) este un concept foarte simplu ce măsoară cât de des apare un anumit termen într-un document, reprezentat de formula $\frac{Nt}{St}$, unde Nt este numărul de apariţii al unui termen într-un text, iar St este numărul total de termeni din document. O practică comună este să normalizăm această valoare prin împărțirea la lungimea în caractere a documentului, deoarece un termen are mai multe șanse să apară mai des într-un document mai lung decât unul mai scurt.

Inversersa frecvenței în document (IDF) măsoară cât de important este un anumit termen. Când calculăm TF, toți termenii sunt considerați la fel de importanți, dar în orice text vom avea o mulțime de prepoziții sau alte cuvinte de legătură ce vor apărea foarte des, deși relevanța acestora va fi limitată. Avem deci nevoie de o metodă ce va da mai multă importanță celor mai rare, iar formula pentru IDF va fi:

$$IDF = \ln\left(\frac{Sd}{N_{dt}}\right)$$

În formula, Sd este numărul total de documente, iar N_{dt} este numărul de documente în care apare termenul t.

Putem să luăm un exemplu în care un document cu 400 de cuvinte conţine cuvântul “lucrare” de 21 de ori. În acest caz, frecvenţa termenului “lucrare” va fi egală cu $\frac{21}{400} = 0.0525$. Dacă avem o mulţime de 10 documente şi cuvântul lucrare apare în 3 dintre acestea, atunci $\ln\left(\frac{10}{3}\right) = 1.2$ deci factorul TF-IDF va fi înmulţirea dintre TD şi IDF, adică 0.063. Dacă vom sorta descrescător rezultatele pentru fiecare cuvânt întâlnit în document, vom putea extrage cele mai relevante cuvinte din textul nostru.

Implementarea în cod a acestui concept este suficient de simplă încât nu a fost nevoie de o librărie specială (spre deosebire de metodele de NLP, Word2Vec sau PageRank).

```
 public static List<Dictionary<string, double>> Transform(List<List<List<string>>> stemmedDocuments,
                                                          int vocabularyThreshold = 3)
 {
1.      Dictionary<string, double> vocabularyIDF = new Dictionary<string, double>();

2.      List<string> vocabulary = GetVocabulary(stemmedDocuments, vocabularyThreshold);

3.      vocabulary = vocabulary.Where(word => word.All(letter => char.IsLetterOrDigit(letter))).ToList();

        foreach (var term in vocabulary)
        {
4.              int numberOfDocsContainingTerm = stemmedDocuments.Where(document =>
                                                          document.Where(sentence =>
                                                          sentence.Any(word =>
                                                          word.ToLower() == term)).Any()).Count();

5.                  vocabularyIDF[term] = Math.Log(stemmedDocuments.Count / numberOfDocsContainingTerm != 0 ?
                                                  (1 + numberOfDocsContainingTerm) : 1);
         }

6.      return TransformToTFIDFVectors(stemmedDocuments, vocabularyIDF);
   }
```

Implementarea a fost făcută în doi pași, unde primul pas este metoda de mai sus, ce are ca parametru o matrice tridimensională deoarece avem o listă de documente, unde fiecare document este o listă de propoziții, iar fiecare propoziție este o listă de cuvinte. La linia 2 vom obține vocabularul documentelor, adică lista de cuvinte ce apar în toate documentele, cu condiția ca fiecare cuvânt să apară de cel puțin 3 ori. Linia 3 filtrează cuvintele, păstrând doar cuvintele alcătuite doar din litere și cifre. Apoi se iterează prin toate cuvintele din vocabular și pentru fiecare se găsește numărul de documente în care apare pentru ca apoi la linia 5 să se umple un dicționar unde cuvântul servește drept cheie și valoarea IDF servește drept valoare.

```
 private static List<Dictionary<string, double>>TransformToTFIDFVectors(List<List<List<string>>> stemmedDocs,
                                                                        Dictionary<string, double>vocabularyIDF)
  {
       List<Dictionary<string, double>> returnList = new List<Dictionary<string, double>>();
       int documentIdx = 0;
       foreach (var doc in stemmedDocs)
       {
               returnList.Add(new Dictionary<string, double>());
               foreach (var vocab in vocabularyIDF)
               {
                   double tf = 0.0f;
1.                 doc.ForEach(sentence => tf += sentence.Where(word => word.ToLower() == vocab.Key).Count());

2.                 double tfidf = tf * vocab.Value;
                   returnList[documentIdx][vocab.Key] = tfidf;
               }
               documentIdx++;
           }
           return returnList;
       }
```

Metoda `TransformToTFIDFVectors` nu face decât să calculeze pentru fiecare document și pentru fiecare termen din vocabular factorul TD*IDF înmulțind valoarea calculată precedent (linia marcată cu 2.) cu frecvența termenului în documentul curent (linia marcată 1.). La final vom avea o listă de dicționare, fiecare dicționar ținând maparile <cuvant, TD*IDF> pentru un document.

1.8 Word2Vec[8][16][17]

În domeniul procesării limbajului natural s-au făcut numeroase eforturi de-a lungul timpului pentru a reprezenta cuvinte și expresii sub o formă vectorială precisă, iar Word2Vec este unul dintre cele mai bune modele pentru această sarcină. Scopul său este de a oferi o valoare vectorială fiecărui cuvânt, dar și de a plasa cuvintele similare în zone vecine din spațiu.

În seria de lucrări [8], [16] și [17] se introduce și se îmbunătățește modelul Skip-gram, o metodă foarte eficientă de a găsi reprezentări vectoriale pentru cuvinte, învățarea făcându-se nesupervizat și fară operații matriceale complexe, lucru care face partea de antrenare extrem de rapidă, modelul fiind capabil să învețe peste 100 de miliarde de cuvinte într-o singură zi.

Cea mai puternică proprietate pe care acest model o are este faptul că putem executa operații liniare pe vectorii oferiți de acesta, iar rezultatul operației păstrează o valoare aceeași semnatică corectă pe care o aveau și operanzii. Exemplul oferit de autori este faptul că operația “vec(Madrid) – vec(Spania) + vec(Franța)” este mai aproape de vectorul vec(Paris) decât de orice alt vector învățat.

Modelul Skip-gram este inspirat dintr-un model clasic de învățare: “Continuous Bag of Words” (CBOW). În CBOW reprezentări diferite ale contextului dintr-o propoziție sunt combinate pentru a prezice un cuvânt din interior. De exemplu, dacă antrenăm modelul cu propoziția: “Lucrarea de dizertație este terminată”.

Dacă vom da ca intrare vectorul de cuvinte [“Lucrarea”, “este”, “terminată”] vom primi ca ieșire cuvântul “dizertație”. Dacă am folosi modelul Skip-gram pentru aceeași propoziție, am putea obține context pe baza unui cuvânt. Astfel, pentru cuvântul “dizertaţie” am primi valori foarte mari în nivelul de ieșire al rețelei neuronale pentru cuvintele “lucrare” și “terminată”.

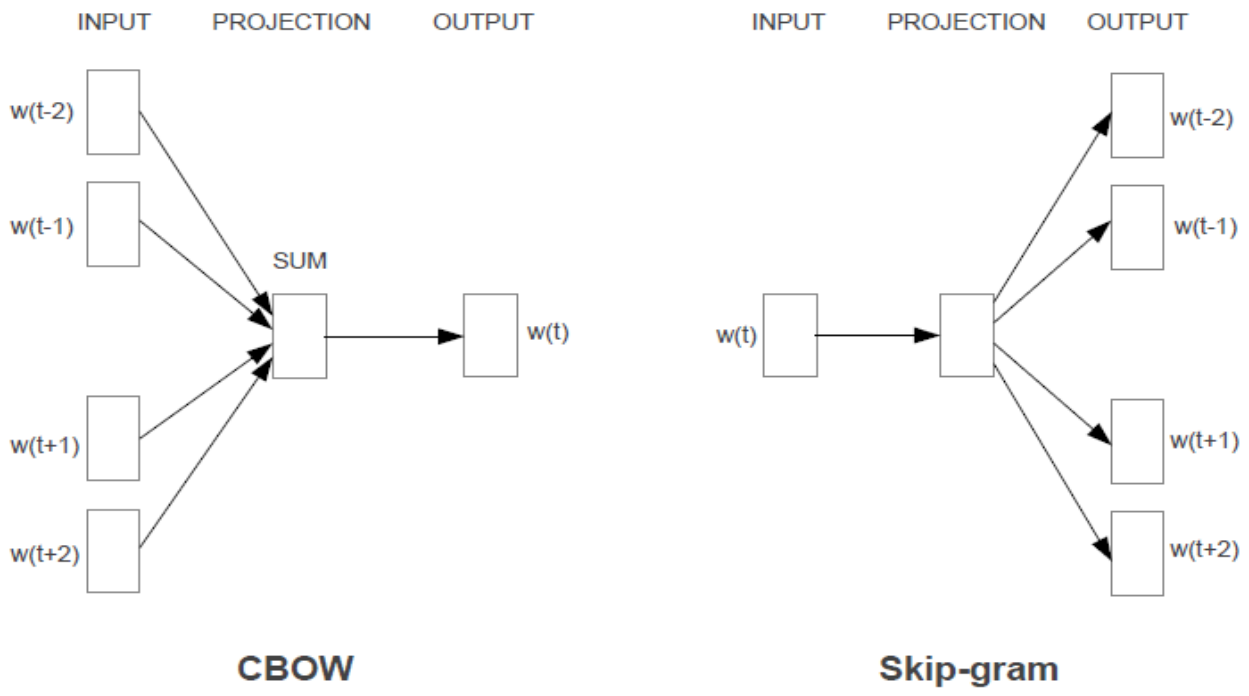

Figura 1.5 Structura reţelei CBOW vs Skip-gram [5]

Putem vedea în figura 1.5 că vectorul de intrare în rețeaua Skip-gram va fi de dimensiunea 1xV, unde V este dimensiunea vocabularului de cuvinte. Nivelul următor este nivelul ascuns ce are un număr de E neuroni, deci acesta va avea dimensiunea VxE. Ieșirea acestui nivel va merge în nivelul final de dimensiune 1xV unde fiecare valoare din vector va reprezenta probabilitatea de a avea cuvântul dat ca intrare pe acea poziție.

Partea de back-propagation pentru mulțimea de antrenare se va face într-un singur pas. După pasul inițial în care ajungem la nivelul final, vom calcula vectorii eroare pentru fiecare cuvânt, deci vectori 1xV. Sumarea acestor vectori element cu element ne va da un singur vector eroare pe baza căruia se pot modifica parametrii (weights) rețelei.

În cazul CBOW, nivelul ascuns și cel de ieșire vor rămâne la fel, dar se va schimba intrarea și activarea rețelei. Dacă avem ca în exemplu anterior, 3 cuvinte de context atunci vom avea 3 vectori 1xV și fiecare va fi înmulțit cu nivelul ascuns VxE pentru a returna 3 vectori 1xE care pot fi uniți printr-o operație de medie aritmetică la nivel de element pentru a oferi activarea finală pentru nivelul de ieșire.

Lucrarea [8], de la aceeași autori, propune o extensie semnificativă asupra modelului Skip-gram. Dacă înainte aveam vectorii vec(Boston) și vec(Globe), nu ne puteam aștepta ca și combinația

lor să ofere un rezultat semantic corect cu respect la ziarul “Boston Globe”. Exemplele compuse precum “Boston Globe” vor fi identificate în partea de antrenare și introduse mai departe că o mapare cu unul din cuvintele de bază, deci “Boston” va avea asociat termenul compus “Boston Globe”.

Un alt avantaj pe care reprezentarea vectorială îl oferă, pe lângă capacitatea de a executa operații liniare, este faptul că putem măsura similaritatea între vectori, deci și între cuvinte, folosind cosinusul unghiului dintre acestea. Metoda de calcul este foarte simplă:

$$cosSimilarity(A,B)=\frac{dotP(A,b)}{|A|*|(B)|}$$

Produsul scalar a doi vectori ne oferă produsul magnitudinii vectorilor înmulțit cu cosinusul unghiului dintre cei doi vectori, deci putem recupera cosinusul împărțind la cele două magnitudini.
Aceste două proprietăți vor fi esențiale în operația de sumarizare a textului din partea a doua a lucrării, dar și dimensiunea impresionantă a modelului Word2Vec va servi drept încă un nivel de filtrare după cel din lucrarea [5] și cel de NLP menționat anterior.

1.9 PageRank[6]

Cel mai cunoscut algoritm folosit în lucrare este algoritmul PageRank. Inventat de Sergey Brin și Lawrence Page acesta a pus bazele motorului de căutare Google, introdus în lucrarea [6]. Caracteristica de bază pe care cei doi o doreau de la motorul lor de căutare era exploatarea structurii de graf a WWW (într-un mod similar cu ce am descris în subcapitolul 1.3) și de a sorta aceste pagini după importanța lor, iar pentru acest scop au construit algoritmul PageRank. Formula de calcul a indexului PageRank pentru o anumită pagină este:

$$PR(A)=(1-d)+d\left(\frac{PR(B)}{C(B)}+\frac{PR(C)}{C(C)}+\frac{PR(D)}{C(D)}+...\right)$$

În această formulă, PR(A) reprezintă indexul PageRank, C(A) reprezintă numărul de URL ce pleacă din pagina A, iar d reprezintă un factor de amortizare pe care îl vom descrie în mai mult detaliu în cele ce urmează. Partea “(1-d)” din formulă se va asigura că suma tuturor paginilor va fi 1, dar va asigura și faptul că o pagină nouă spre care nici o altă pagină nu are un link va avea un index de (1-d).

Formula de calcul a indexului PageRank poate ridica semne de întrebare la prima vedere din cauza faptului că indexul fiecărei pagini este calculat în funcție de indexul altor pagini. Totuși, eleganța algoritmul este datorată faptului că putem pleca de la orice valoare inițială cu indexul paginilor și apoi trebuie doar să iterăm conform formulei până când paginile vor avea un index ce va varia foarte puțin.

Cea mai simplă și mai intuitivă descriere a algoritmului este din ipoteza unei persoane ce se uită pe o pagină web și după un timp, alege aleatoriu un URL din pagina curentă pentru a merge pe o altă pagină. Probabilitatea ca această persoană să ajungă o anumită pagina vă fi indexul PageRank pentru pagina respectivă. Factorul de amortizare d este probabilitatea la fiecare pagină ca persoana să se plictisească și să meargă pe o altă pagină aleatorie. Acest factor permite un nivel de personalizare al algoritmului, autorii folosindu-l cel mai des cu valoarea de 0.85.

O altă justificare intuitivă oferită de autori este simplul fapt că formula va oferi un index PageRank mare unei pagini spre care duc multe alte pagini, sau puține pagini care au la rândul lor un index foarte mare. Chiar o singură legătură cu o pagină de pe un website foarte cunoscut este suficientă pentru a da relevanță unei noi pagini (de exemplu, un nou articol de pe un site precum cel al BBCului).
Ca un exemplu simplu, vom lua o pagină A care are un URL către pagina B și o pagină B care are un URL către pagina A. Intuitiv, indexul ambelor pagini ar trebui să fie 1.
Dacă aplicăm formula (cu d = 0.85), acest lucru va fi confirmat:

$$PR(A)=(1-d)+d(\frac{PR(B)}{1}) \qquad PR(B)=(1-d)+d(\frac{PR(A)}{1})$$

$$PR(A)=0.15+0.85*1 \qquad PR(B)=0.15+0.85*1$$

Dacă vom încerca să setăm valorile inițiale drept un număr aleator precum 40 vom obține:

$$PR(A)=0.15+0.85*40=34.25 \qquad PR(B)=0.15+0.85*34.25=29.1775$$

$$PR(A)=0.15+0.85*29.1775=24.95 \qquad PR(B)=0.15+0.85*24.95=21.3575$$

$$PR(A)=0.15+0.85*21.3575=18.303 \qquad PR(B)=0.15+0.85*18.303=15.708$$

Dacă am continua iterațiile s-ar observa că valorile PR(A) și PR(B) vor scădea până când se vor stabiliza în jurul valorii 1.0, deoarece PageRank formează o distribuție de probabilitate normalizată.

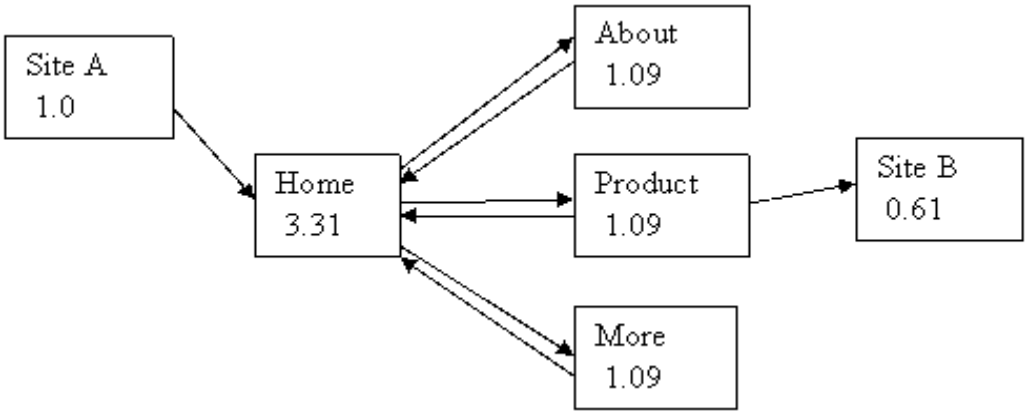

Figura 1.6. O topologie de website uzuală şi cotele PageRank pentru paginile comune. [6]

Figura 1.6 ne arată faptul că intuiția algoritmului se confirmă, iar pagina principală are cel mai mare index, deoarece numeroase alte pagini importante precum pagina de descriere sau orice pagină de produs vor avea URL-uri către aceasta. Tot acest exemplu confirmă și euristica din lucrarea [4] referitoare la pagina principală și graful important pe care aceasta îl creează cu paginile de care se leagă și care se întorc înapoi la aceasta.

1.11 TextRank[7]

Având o familia de algoritmi PageRank ce sunt capabili să ofere un index de relevanță nodurilor din graf, Rada Mihalcea și Paul Tarau au adaptat aceste concepte pe grafuri lexicale sau semantice extrase din documente, reușind să folosească informația extrasă din întregul text pentru a lua decizii de selecție la nivel de propoziție sau cuvânt.

Autorii păstrează cu exactitate formula folosită de Larry Page și Sergey Brin pentru indexul PageRank al unui nod, păstrând și factorul de amortizare. Scorurile inițiale pot fi valori aleatoare pentru că după un număr de iterații vor converge către valorile reale, după cum am demonstrat în subcapitolul anterior. Majoritatea grafurilor obținute din documente vor fi totuși grafuri neorientate, neavand direcția dată de URL-uri în cazul paginilor web. De asemenea este nevoie și de aplicarea unor parametri (weights) pe legăturile dintre noduri.

Astfel, se introduce o formulă parţial modificată faţă de cea din lucrarea inţială:

$$WS(V_i) = (1-d) + d * \sum_{V_j \in In(V_i)} \frac{w_{ji}}{\sum_{V_k \in Out(V_j)} w_{jk}} WS(V_j)$$

Formula TextRank, bazată pe PageRank [7]

Spre deosebire de cazul uzual, în care indexul unui nod este suma indecșilor vecinilor, acum se adaugă și o pondere a "puterii" legăturilor împărțită la suma ponderilor.

Cum am menționat, autorii introduc o întreagă familie de algoritmi, pentru că multe elemente din metoda acestora sunt parametrizabile: putem considera nodurile din graf ca fiind cuvinte, propoziții, paragrafe sau orice altă structură dintr-un document, putem alege un alt algoritm diferit de PageRank pentru a calcula indecșii nodurilor. De asemenea putem alege metoda de a calcula parametrii de pe legături. O descriere abstractă a metodei propuse este:

1. Identificarea unităţilor de text ce vor reprezenta nodurile din graf.
2. Identificarea relaţiilor dintre noduri, dar şi tipul grafului (orientat, neorientat, cu legături cu cost sau nu).
3. Alegerea algoritmului de indexare a nodurilor.
4. Sortarea nodurilor şi alegerea unei submulţimi de noduri.

Autorii oferă un exemplu foarte interesant pentru extragerea cuvintelor cheie (o alternativă foarte bună la TF-IDF). Cuvintele reprezintă nodurile din graf, iar relația dintre acestea va fi următoarea: două noduri sunt conectate dacă acestea apar la distanța de N cuvinte în text, unde N poate fi setat oriunde între 2 și 10. Desigur, se pot adăuga și alte restricții precum faptul că doar substantivele și verbele să poată fi reprezentate în graf, lucru posibil datorită etichetării cuvintelor cu partea de vorbire. Se pot prelua și sintagme de mai mult de un singur cuvânt, dar dimensiuna grafului va crește semnificativ.

Keywords assigned by TextRank:
linear constraints; linear diophantine equations; natural numbers; nonstrict inequations; strict inequations; upper bounds

Keywords assigned by human annotators:
linear constraints; linear diophantine equations; minimal generating sets; non–strict inequations; set of natural numbers; strict inequations; upper bounds

Figura 1.7 Cuvintele cheie alese de PageRank versus cuvintele cheie alese de operatori umani [8]

În imaginea din lucrare se pot vedea cele mai importante cuvinte identificate de TextRank și o selecție de cuvinte importante realizată de un actor uman.

A doua aplicație interesantă a algoritmului TextRank este cazul propozițiilor cheie dintr-un text. Nodurile din graf vor fi propoziții, dar relația folosită în cazul cuvintelor nu poate fi păstrată, deci o noua relație este introdusă de către autori, cea de similaritate. Similaritatea a două propoziții poate fi pur simplu numărul de cuvinte comune dintre cele două. Pentru a nu favoriza propozițiile lungi, autorii normalizează rezultatele împărțind numărul de cuvinte comune dintre doua propoziții la lungimea propozițiilor, oferindu-ne formula:

$$Similarity(S_i, S_j) = \frac{|\{w_k | w_k \in S_i \& w_k \in S_j\}|}{log(|S_i|) + log(|S_j|)}$$

Formula pentru indicele de similaritate între doua propoziții [9]

În concluzie, TextRank are avantajul de a extrage conținutul relevant bazându-se exclusiv pe textul dat ca intrare. De asemenea, spre deosebire de alți algoritmi de învățare supervizată, TextRank este complet nesupervizat și se bazează doar pe text pentru a realiza un rezumat extractiv al textului (o submulțime cu cele mai relevante propoziții).

CAPITOLUL 2 – Implementarea

2.1 Limbajul de programare utilizat [18]

Limbajul ales pentru dezvoltarea aplicației este limbajul C#, realizat de Anders Heljsberg și Scott Wiltamuth, anunțat și lansat de către Microsoft în Iunie 2000, împreună cu platforma .NET. C# este un limbaj orientat pe obiecte proiectat pentru a oferi un amestec optim de simplitate, expresivitate și performanță. Autorii limbajului au avut avantajul de a cunoaște experiența altori limbaje, mai ales Java și C++. Funcționalitatea de bază a aplicației poate fi exportată ca o librărie și datorită sistemului .NET Core, aceasta va putea rula și pe Linux, deși va rula fară interfața grafică.

Cele mai importante și relevante caracteristici pentru care C# a fost limbajul ales pentru lucrare sunt:

- Language Integrated Query (LINQ) – lansat cu C# 3.0, LINQ este un limbaj de interogare foarte puternic și expresiv, inspirat din Structured Query Language (SQL) și foarte util pentru a procesa date stocate în colecții. Vom folosi foarte mult LINQ pentru a programa într-un stil similar cu stilul funcțional de programare, încercând să folosim cât mai multe funcții ce returnează către alte funcții și să folosim funcțiile de LINQ în loc de funcțiile noastre pentru procesarea datelor.
- Parametri opționali – foarte utili pentru clasele cu foarte multe metode, scăpându-ne de nevoie a scrie mult mai multe supraîncărcări ale metodelor curente.
- Operatorul condițional de NULL – operator ce scapă de foarte multe blocuri condiționale în care trebuie să verificăm în mod repetat obiectele pe care facem operații pentru a ne asigura că acestea nu au valoarea NULL. De exemplu: “var object = anotherObject?.object;”. Operatorul “?.” ne scapă de a mai verifica cu un bloc “if” existența obiectului “anotherObject”.
- Suport pentru paradigma de programare funcțională – C# oferă suport pentru funcții exprimate ca obiecte (Func<int, bool>) fiind o funcție ce ia ca parametru un număr întreg și oferă ca ieșire o valoare booleană), funcțiile pot lua ca parametru și pot returna alte funcții, funcțiile pure (ce nu cauzează efecte secundare) pot fi exprimate prin metode statice, suport pentru funcții anonime.
- Suport extrem de bun pentru programarea paralelă, în special programarea paralelă asincronă, extrem de importantă în cazul nostru datorită naturii aplicației. Avem nevoie să mutăm calculele complexe de pe firul de execuție principal pentru ca acesta să se ocupe strict de mesajele de la sistemul de operare și a întreține interfața grafică.

Alte alternative ce au fost luate în calcul:

- Python – limbajul cel mai des întâlnit în sfera învățării automate și a procesării limbajului natural. Acest limbaj oferă simplitatea prezentă în C#, dar nu are unele funcționalități care ușurează foarte mult dezvoltarea unei aplicații bazate pe colecții, vorbind în special despre LINQ. Un avantaj al limbajului Python este faptul că există foarte multe librării pentru probleme abordate în lucrare, dar experiența a dovedit că se găsesc alternative sau chiar rescrieri ale acelor librării și pentru C#.
- C++ - un avantaj incontestabil al limbajului C++ este viteza de execuție. Aplicația efectuează calcule complexe pe seturi largi de date, deci C++ ar fi oferit o viteză mai mare de execuție, dar costul ar fi fost timpul de dezvoltare mărit din cauza responsabilităților prezente în scrierea de cod C++ față de C#. Dezavantajul este că lipsa de librării este mult mai pronunțată în cazul C++, deși acesta este folosit foarte des în librăriile de învățare automată pentru executarea operațiilor matriceale. Totuși, librăriile nu expun foarte des codul C++, acesta fiind chemat de limbaje precum Python sau C#.

2.2 Sistemul utilizat pentru crearea interfeţei grafice [19][20]

Cum am menționat, chiar dacă aplicația este scrisă în C#, asociat în mod normal cu sistemul de operare Windows, Microsoft a lucrat la sistemul .NET Core ce permite execuția de cod scris în C# și pe sistemul Linux. Aplicația ar putea rula și doar din linia de comandă, dar pentru că avem un număr de parametri pe care trebuie să îi dăm aplicației, este mai ușor să avem un sistem grafic în care să putem introduce valorile necesare și să vedem starea procesului curent, deoarece timpul de execuție este destul de lung în cazul multor funcționalități. Pentru acest scop am ales Windows Presentation Foundtation (WPF) datorită ușurinței de a dezvolta interfețe grafice, WPF fiind bazat pe formatul Extensible Application Markup Language (XAML).

XAML împrumută caracteristici din HTML și oferă un mod simplu de definire a controalelor grafice necesare. Putem vedea un exemplu cu un container și un buton:

```
<StackPanel>
        <Button Content="Click Me"/>
</StackPanel>
```

WPF foloseşte o structură de arbore pentru reprezentarea interfeței, deci rădăcina arborelui va fi mereu un prim container care va conține oricâte alte containere ce pot conține toată suita de controale grafice oferite de WPF (butoane, radio-butoane, casete text etc).

2.2 Extragerea şablonului paginilor cu conţinut

În arhitectura aleasă pentru acest proiect, websiteul din care vom extrage informaţie este o clasă numită DynamicallyScrapedWebsite ce se va ocupa de extragerea șablonului și de regăsirea informației, pe când o altă clasă se va ocupa de sumarizare (pentru a respecta regulile POO prin care o clasă trebuie să se ocupe de un singur lucru). Cum am menționat, singurul parametru real de intrare în program este URLul către pagina principala a site-ului. Toată partea de extragere se execută într-o singură metodă ce se execută pe alt fir de execuție în mod asincron (pentru a nu bloca firul de execuție principal). Vom lua analiza codul metodei în mai multe părți deoarece aceasta este destul de lungă și conține toate conceptele teoretice din subcapitolele 1-7.

```
1. var mainPageDocument = await GetDocumentFromLink(siteUrl).ConfigureAwait(true);

2. var mainPageLinksStrings = mainPageDocument.Links.Where(webPageLink =>
                                (webPageLink as IHtmlAnchorElement)?.Href != siteUrl)
                                .ToList();
```

DOMul este obţinut cu ajutorul librăriei AngleSharp, peste care am făcut o altă metodă asincronă ajutătoare. Metoda GetDocumentFromLink este asincronă deoarece aceasta trebuie să facă o cere HTTP de tip GET către serverul ce ține pagina respectivă, deci răspunsul este asincron, iar noi trebuie să blocăm execuția programul cu “await” până când funcția aceasta returnează structura de DOM.

La linia 2 obţinem toate URLurile din pagină folosind proprietatea Links de pe obiectul de DOM, dar folosim o interogare LINQ de tip “Where” pentru cazul în care pagina principală conţine un URL către ea însăşi (nu vrem să avem noduri în graf care au legături către ele însele).

```
1.  HashSet<LinkToBeProcessed> mainPageLinks = new HashSet<LinkToBeProcessed>();
2.  mainPageLinksStrings.ForEach(mainPageLinkString =>
    mainPageLinks.Add(new LinkToBeProcessed(mainPageLinkString, siteUrl, 1)));
3.  RefreshLinksHashSet(ref mainPageLinks);
```

Vom ține URLuri pe care le vom procesa într-o mulțime (HashSet), pentru a evita duplicatele. Clasa LinkToBeProcessed ține 3 membri: URLul în sine, URLul către pagina principală și un număr întreg ce denotă prioritatea cu care acesta trebuie să fie procesat (vom vorbi de prioritate mai târziu când vom ajunge la extragerea de informație în sine).

Funcția RefreshLinksHashSet este un pas de optimizare foarte important adăugat în lucrare la algoritmul de extragere din [4]. Procesarea unui URL implică operații foarte costisitoare: descărcarea și construirea DOMului (asincronă și dependentă de rețea), dar și adăugarea în graf și căutarea unui nou subgraf. Din acest motiv, adaugăm 3 euristici pentru a elimina 2 clase de URLuri și de a optimiza căutarea:

```
   private void RefreshLinksHashSet(ref HashSet<LinkToBeProcessed> linksHashSet)
   {
1.     linksHashSet = linksHashSet.Where(linkToProcess => linkToProcess.link !=
                                    "javascript:void(0)").ToHashSet();

2.     linksHashSet = linksHashSet.Where(linkToProcess => linkToProcess.link.Length >
                                                    siteUrl.Length).ToHashSet();

3.     linksHashSet = linksHashSet.OrderByDescending(linkToProcess =>
                                linkToProcess.priority).ThenByDescending(linkToProcess =>
                                linkToProcess.link.Length).ToHashSet();
   }
```

Linia 1 elimină toate URLurile ce nu sunt implementate complet și nu duc nicăieri. Apoi la linia 2 se elimină toate URLurile ce sunt mai scurte decât URLul paginii principale. Există câteva cazuri extrem de rare în care acest lucru poate elimina URLuri relevante, dar de cele mai multe ori o pagină de conținut va fi rezultatul concatenării URLului către pagina principală și un identificator unic extras din titlu. De exemplu, siteul BBC rezidă la adresa "***https://www.bbc.com/***", iar unul din articolele sale are URL-ul de forma "***https://www.bbc.com/news/world-us-canada-52428994***". Astfel, nu trebuie să procesăm URLuri mai scurte decât URLul paginii de start. O ultima euristică va fi faptul că vom sorta URLurile din mulțime descrescător după lungime. Folosind același argument că la linia 2, URLuri mai lungi duc de obicei către pagini cu informație mult mai specifică. Această funcție va fi apelată de mai multe ori în situații diferite pentru a optimiza URLurile pe care vom merge.

Fiind la pasul la care trebuie să găsim un subgraf alcătuit din pagini la care putem ajunge pornind de la pagina principală (și care conțin și URLul de întoarcere către pagina principală), trebuie să elaborăm codul ce se ocupă de această parte (implementarea algoritmului din lucrarea [4]):

```
1. var webPage = await GetSubPageFromLink(link).ConfigureAwait(true);
   if (webPage == null)
        continue;

2. processedLinks.Add(link);

3. var existingLinks = webPage.Links.Where(webPageLink =>
                                          (webPageLink as IHtmlAnchorElement)?.Href != link)
                                          .Select(webPageFilteredLink =>
                                          (webPageFilteredLink as IHtmlAnchorElement)?.Href)
                                          .ToList().Intersect(mainPageLinks).ToList();

4.  if (existingLinks.Count <= 1)
        continue;

5.  existingLinks.ForEach(webPageLink => connections.Add(new Connection<string>(link,
                                                                                webPageLink)));

6.  List<HashSet<string>> allCompleteSubdigraphs = GetAllCompleteSubdigraphs(connections);

7.  if (connections.Count > MaxConnectionsCount && testedGraphs.Count > 0)
        break;
```

Pentru fiecare URL de pe pagina principală, vom descărca și construi conținutul DOM, având grijă să marcăm URLul ca și procesat (sau vizitat) pentru a nu reveni asupra lui dacă acesta apare de mai multe ori în pagina principală. Linia 3 se ocupă de multe lucruri, folosind o înșiruire de operații LINQ: luăm toate URLurile de pe pagină pe care o procesăm în mod curent, având grijă să nu avem cazul în care o pagina conține un URL către ea însăși, apoi vom lua intersecția dintre aceasta mulțime de URLuri și mulțimea celor din pagina principală(această intersecție reprezintă URLurile dintre pagina principală și pagina curentă, ținta noastră fiind ca ea să reprezinte meniul principal).

Graful întregului site este ținut sub forma unei liste de conexiuni, un obiect de tip conexiune având 2 membri de tip șir de caracter, reprezentând paginile ce sunt legate printr-un URL. La linia 5 adaugăm conexiunile găsite la această iterație pentru ca apoi să calculăm toate subgrafurile complete existente în graf. Linia 7 reprezintă doar un caz de oprire deoarece unele siteuri pot fi atât de mari încât căutarea să dureze mult prea mult. Când graful siteului la momentul curent ajunge să conțină un număr mai mare de legături decât cel dat că parametru(vom putea da acest parametru din interfață) ne vom opri și vom alege cel mai bun subgraf întâlnit până în acest moment(urmează să descriem modul în care subgrafurile sunt sortate.)

Problema găsirii tuturor subgrafurilor complete dintr-un graf este dificilă, neavând un algoritm ce poate rezolva problema în timp polinomial(problema este "NP-hard"). Lucrarea [4] nu menționează algoritmul folosit pentru această etapă. Cel mai cunoscut algoritm pentru această sarcină ar fi algoritmul Bron-Kerbosch, dar pentru a ține implementarea la un nivel mai minimalist la prima iterație, a fost folosit un algoritm mult mai brut.

```
            foreach (var pLink in processedLinks)
            {
                HashSet<string> newCS = new HashSet<string>();
                foreach (var connectionOne in connections)
                {
                    if (connectionOne.end1 == pLink)
                    {
                        foreach (var connectionTwo in connections)
                        {
                            if (connectionTwo.end2 == pLink &&
                                connectionTwo.end1 == connectionOne.end2)
                            {
                                newCS.Add(connectionTwo.end1);
                                newCS.Add(connectionTwo.end2);
                            }
                        }
                    }
                    else if (connectionOne.end2 == pLink)
                    {
                        foreach (var connectionTwo in connections)
                        {
                            if (connectionTwo.end1 == pLink &&
                                connectionTwo.end2 == connectionOne.end1)
                            {
                                newCS.Add(connectionTwo.end1);
                                newCS.Add(connectionTwo.end2);
                            }
                        }
                    }
                }

                if (!allCompleteSubdigraphs.Any(cs => cs.SetEquals(newCS)) && newCS.Count >=
MaximalSubdigraphSize)
                    allCompleteSubdigraphs.Add(newCS);
            }
```

Pornind de la un URL din mulțimea de URLuri procesate, trebuie să găsim o conexiune din graf ce are URL-uri respectiv la un capăt(având una dintre direcții). Apoi trebuie să găsim și conexiunea ce reprezintă legătura în cealaltă direcție din graf, deci o conexiune ce are URLul curent la celalalt capăt. Din păcate, acest algoritm este alcătuit din 3 bucle, deci rulează în complexitate temporală de N*C*C, unde este N este numărul de URLuri procesate, iar C este numărul de conexiuni din graf. Pe siteuri

foarte mari, ce conțin sute de URL-uri pe fiecare pagină, acest algoritm va converge foarte greu către un subgraf optim din care să putem extrage șablonul.

Suntem la pasul în care trebuie să testăm fiecare subgraf găsit pentru a vedea cât de asemănătoare(din punct de vedere al elementelor HTML cuprinse) sunt paginile din acesta. După cum am menționat în introducerea teoretică, aici vom folosi o abordare proprie, diferită de cea din lucrarea [4](compararea sus-jos a arborilor).

```
1.    foreach (var page in bestCS)
      {
             var webPageCS = await GetSubPageFromLink(page).ConfigureAwait(true);
              if (webPage == null)
                  continue;
              webDocuments.Add(webPageCS);
      }


2.    var pagesFrequencyDictionaryList = webDocuments.Select(dom => dom.All.GroupBy(element =>
                element.GetType().ToString()).ToDictionary(x => x.Key, x => x.Count())).ToList();

3.    double averageStandardDeviation = GetSimilarityBetweenTemplates(pagesFrequencyDictionaryList);
      if (averageStandardDeviation > 0 && averageStandardDeviation < thresholdStandardDevianceTemplate)
      {
              foundSubdigraph = true;
              break;
      }
      else
      {
4.            testedGraphs.Add(new Tuple<double, HashSet<string>, List<IHtmlDocument>, Dictionary<string, int>>
                              (averageStandardDeviation, new HashSet<string>(bestCS), new
                               List<IHtmlDocument>(webDocuments), new Dictionary<string, int>(websiteTemplate)));

      }
```

Linia 1 pur și simplu descarcă DOMul pentru fiecare pagină din subgraf. Acest pas este de fapt costisitor din motivele pe care le-am mai menționat(lucrul cu rețeaua și faptul că trebuie să avem aceste cereri pe rețea cu directive “await”, în mod blocant). Dacă vom testa multe subgrafuri, timpul de căutare va crește extrem de mult, iar acest pas nu poate fi optimizat algoritmic.

Linia 2 va construi vectorii de frecvență pentru toate paginile dintr-un singur pas. Structura va fi una de dicționar, unde cheia va fi eticheta HTML a unui tip de element, iar valoarea va fi numărul de apariții ale acesteia în DOMul respectiv.

Linia 3 va calcula indexul de similaritate al subgrafului curent în modul prezentat în teorie, deși vom prezenta și codul metodei de calcul mai jos. Funcția primește orice mulțime de șabloane(în cazul nostru, presupunem că fiecare pagină din subgraf ar putea fi un șablon) și returnează similaritatea dintre acestea, dar va construi de asemenea și combinația șabloanelor într-unul singur. Dacă acest index este

mai mic decât o valoare aleasă de utilizator atunci acest graf va și ales și șablonul va rămâne cel identificat la acest pas, dacă nu, vom reține oricum informații despre acest graf(un tuplu cu indexul obținut, subgraful în sine, lista de DOMuri și șablonul obținut din acest subgraf).

Funcția GetSimilarityBetweenTemplates are următorul comportament:

```
        HashSet<string> allHTMLTags = new HashSet<string>();
1.      mainAndCurrentPageTemplates.ForEach(pageFrequencyDictionary =>
                    pageFrequencyDictionary.Keys.ToList().ForEach(tag => allHTMLTags.Add(tag)));

2.      foreach (var pageDictionary in mainAndCurrentPageTemplates)
        {
            foreach (var tag in allHTMLTags)
            {
                if (!pageDictionary.Keys.Contains(tag))
                    pageDictionary[tag] = 0;
            }
        }

        Dictionary<string, double> templateStandardDeviation = new Dictionary<string, double>();

        var tagsArray = allHTMLTags.ToArray();
3.      for (int iTagIdx = 0; iTagIdx < allHTMLTags.Count; iTagIdx++)
        {
            List<int> tagValues = new List<int>();
4.          foreach (var pageDictionary in mainAndCurrentPageTemplates)
                tagValues.Add(pageDictionary[tagsArray[iTagIdx]]);

5.          websiteTemplate[tagsArray[iTagIdx]] = tagValues.GetMedian();
6.          templateStandardDeviation[tagsArray[iTagIdx]] = tagValues.GetStandardDeviation();
        }

7.      return templateStandardDeviation.Values.Sum() / templateStandardDeviation.Values.Count;
```

La linia 1 se construiește o mulțime(pentru a ignora duplicatele) ce conține toate etichetele din toate paginile din subgraf. Această mulțime va fi folosită la linia 2 pentru a completa fiecare vector de frecvență cu etichetele ce nu apar în acesta(vrem să facem operații cu vectori de aceeași mărime).

Apoi în bucla de la linia 3 vor itera prin toate etichetele și vom strânge numărul de apariții din fiecare vector(linia 4) pentru ca în final să obținem mediana(ce va servi drept valoarea elementului în șablonul final) și deviația standard pentru acest element. La final vom întoarce ca rezultat media aritmetica a deviației standard pentru fiecare etichetă, aceasta fiiind indexul ce descrie similaritatea paginilor din șablon. Dacă acest index corespunde, se va folosi șablonul alcătuit din valorile mediane(se preferă mediana peste media aritmetică pentru a ne proteja împotriva valorilor foarte mari sau mici pe anumite pagini).

2.3 Filtrarea conţinutului

Partea de filtrare a conținutului preluată din lucrarea [5] este destul de intuitiv de implementat. Densitatea de URLuri și densitatea de text se pot calcula ușor la nivel de nod. Modificarea importantă adusă este înlăturarea pasului de fuziune de noduri, din motivul atins în prezentarea teoriei. O a doua modificare importantă adusă este înlăturarea valorilor de prag pentru cei doi factori cu mediile acestora la nivelul documentului.

```
    List<Tuple<AngleSharp.Dom.IElement, float, float>> documentFeatureAnalyis =
              new List<Tuple<AngleSharp.Dom.IElement, float, float>>();

    for (int iNodeIdx = 0; iNodeIdx < document.Count; iNodeIdx++)
    {
         var node = document[iNodeIdx];

         if (node.BaseUrl.Href.Contains("about") && node.BaseUrl.Href.Contains("blank"))
             node.BaseUrl.Href = string.Empty;

1.       documentFeatureAnalyis.Add(new Tuple<AngleSharp.Dom.IElement, float, float>
              (node, node.GetNodeTextDensity(), node.GetNodeHyperlinkDensity()));
    }

2.  float textDensityThreshold = documentFeatureAnalyis.Average(feature => feature.Item2);
3.  float hyperLinkDensityThreshold = documentFeatureAnalyis.Average(feature => feature.Item3);
```

Pentru fiecare nod din DOM vom crea un tuplu ce conține densitatea de text la nivel de nod și densitatea de URLuri la nivel de nod, iar avea vom folosi din nou o funcție LINQ pentru a calcula media aritmetică a acestor valori la nivelul întregului document. Funcțiile de calcul pentru cei doi factori sunt destul de simple deoarece folosesc membri expuși de librăria AngleSharp:

```
    public static float GetNodeTextDensity(this AngleSharp.Dom.IElement element)
    {
        if (element == null)
            return 0.0f;

        return ((float)element.TextContent.Length) / ((float)element.InnerHtml.Length);
    }

    public static float GetNodeHyperlinkDensity(this AngleSharp.Dom.IElement element)
    {
        if (element == null)
            return 0.0f;

        return ((float)element.BaseUri.Length) / ((float)element.InnerHtml.Length);
    }
```

Pentru a îmbunătăți filtrarea, am profitat de faptul că avem informația text la nivel de cuvânt și propoziție datorită segmentării textului pentru a aplica alte euristici. Una din acestea va elimina toate propozițiile în care nici un cuvânt nu a fost etichetat drept verb:

```
    documentResult.scrapingResults.ForEach(element => documentResult.content +=
                                        element.element.TextContent + ".");

1.  var sentences = OpenNLP.APIOpenNLP.SplitSentences(documentResult.content);

    List<string> filteredSentences = new List<string>();
    List<List<string>> sentencesWords = new List<List<string>>();
    foreach (var sentence in sentences)
    {
         var filteredSentence = sentence.Replace("\n", "");
         filteredSentence = filteredSentence.Replace("\t", "");
         filteredSentences.Add(filteredSentence);

2.       sentencesWords.Add(OpenNLP.APIOpenNLP.TokenizeSentence(filteredSentence).ToList());
    }

    List<List<string>> posSentences = new List<List<string>>();
    foreach (var sentenceWordList in sentencesWords)
3.        posSentences.Add(OpenNLP.APIOpenNLP.PosTagTokens(sentenceWordList.ToArray()).ToList());

    List<int> indexesToRemove = new List<int>();
    for (int sentenceIndex = 0; sentenceIndex < posSentences.Count; sentenceIndex++)
    {
        if (!posSentences[sentenceIndex].Any(pos => pos.Contains("V")) ||
             sentencesWords[sentenceIndex].Any(word => word.Length > maximalWordCount))
        {
4.           indexesToRemove.Add(sentenceIndex);
        }
    }
```

La acest pas se vor rula pentru prima oară metodele de NLP din librăria OpenNLP. Deoarece rularea acestora necesită mai mulți pași(precum încărcarea modelului antrenat pe limba engleză) vom avea o clasă înfășurătoare APIOpenNLP ce va simplifica cât mai mult utilizarea librăriei.

La linia 1 vom aplica metoda de segmentare a textului în propoziții(cu o linie înainte se agrega tot textul din nodurile HTML într-un singur șir de caractere) pentru a primi ca ieșire un vector de propoziții prin care vom itera pentru ca la linia 2 să realizăm și segmentarea propozițiilor în cuvinte pentru ca într-un final să etichetăm fiecare cuvânt din fiecare propoziție.

Tipul de conținut ce scapă cel mai des filtrării este codul JavaScript. Acest nivel de filtrare pe baza părții de propoziție este foarte relevant deoarece vom avea foarte rar un caz în care un cuvânt cheie din sintaxa JavaScript sau un nume de variabilă va fi etichetat drept verb. Astfel, mult conținut JavaScript va fi reprezentat de propoziții fară verbe, ce pot fi eliminate (linia 4).

Un ultim nivel de filtrare de bazează pe metoda Word2Vec. Filtrarea nu este scopul principal al acesteia în lucrare, dar la fel cum am folosit partea de NLP (menită extragerii cuvintelor cheie) pentru filtrare, la fel vom folosi Word2Vec (utilizat în principal să ofere indicele de similaritate pentru propozițiile din text) pentru a construi un nou nivel de filtrare.

```
        do
        {
             wordRemovalConverged = true;
             for (int sentenceIndex = 0; sentenceIndex < sentencesWords.Count; sentenceIndex++)
             {
                   List<bool> wordsFoundInSentence = new List<bool>();
                   for (int wordIndex = 0; wordIndex < sentencesWords[sentenceIndex].Count; wordIndex++)
                   {
                        var vec = Word2VecManager.GetVecForWord(sentencesWords[sentenceIndex][wordIndex],
                                                                Word2VecMaxCount);
                         if (vec.Length == 0)
                             wordsFoundInSentence.Add(false);
                         else
                             wordsFoundInSentence.Add(true);
                        }

                        for (int wordIndex = 1; wordIndex < sentencesWords[sentenceIndex].Count - 1; wordIndex++)
                        {
                            if (!wordsFoundInSentence[wordIndex - 1] && !wordsFoundInSentence[wordIndex] &&
                                 !wordsFoundInSentence[wordIndex + 1])
                            {
                                sentencesWords[sentenceIndex].RemoveAt(wordIndex + 1);
                                posSentences[sentenceIndex].RemoveAt(wordIndex + 1);
                                wordsFoundInSentence.RemoveAt(wordIndex + 1);

                                sentencesWords[sentenceIndex].RemoveAt(wordIndex);
                                posSentences[sentenceIndex].RemoveAt(wordIndex);
                                wordsFoundInSentence.RemoveAt(wordIndex);
                                wordIndex--;

                                sentencesWords[sentenceIndex].RemoveAt(wordIndex);
                                posSentences[sentenceIndex].RemoveAt(wordIndex);
                                wordsFoundInSentence.RemoveAt(wordIndex);

                                wordRemovalConverged = false;
                            }
                        }
                    }
                } while (!wordRemovalConverged);
```

Ideea de bază a filtrării pe baza vectorilor din Word2Vec este prezența cuvintelor din textul nostru în baza de cuvinte cunoscută de modelul folosit. Modelul este unul antrenat de Google ce cunoaște 100 de miliarde cuvinte, fiecare cuvânt fiind un vector de 300 de valori reale. Deoarece Google a antrenat acest model pe text de pe internet, acesta include și nume proprii și chiar și foarte multe greșeli de scriere (ex: “Califrnia” în loc de “California”). Totuși, pentru a nu face o filtrare mult prea strictă ce va elimina cuvinte relevante (în general, nume proprii de care nu a auzit deoarece acestea sunt apărute după antrenarea modelului de la Google) vom avea o fereastră de dimensiune 3, adică ne vom uita și la cuvântul din urmă și la cel din față și vom elimina astfel de grupuri de câte 3 cuvinte ce nu sunt găsite în Word2Vec. Deoarece o astfel de iterație va elimina unele grupuri de cuvinte inexistente, eliminarea lor poate crea alte grupuri, deci acest algoritm trebuie rulat până când ajunge la convergență, adică momentul în care nu avem 3 cuvinte consecutive ce nu există în baza de cuvinte încărcată în memorie.

Word2VecManager este clasa înfășurătoare ce se ocupă de încărcarea și utilizarea modelului de la Google. Sunt mai multe lucruri de menționat în legătură cu lucrul cu acest model. Primul este faptul că nu putem încărca realist întreaga baza Word2Vec de pe disc în memoria RAM(Random Acces Memory) deoarece fiecare cuvânt ocupă 1200 de bytes(o variabilă de tip “float” ocupând 4 bytes, iar un cuvânt fiind un vector de dimensiune 300). Astfel, numărul de cuvinte ce vor fi încărcate este parametrizabil prin interfața grafică, dar în general, câteva sute de mii de cuvinte sunt mai multe decât suficiente pentru filtare. Al doilea lucru demn de menționat este direct legat de primul și se referă la faptul că încărcarea în RAM a unui număr mare de cuvinte va dura mult timp.

2.4 Testarea modulului de extragere şi filtrare a conţinutului

O metodă bună de testare pentru partea de extragere și filtrare a conținutul este distanța Levenshtein, cunoscută și sub numele de “edit-distance”. Acest indice de tip întreg reprezintă numărul de schimbări la nivel de caracter(schimbare, șterge, adăugare) pe care ar trebui să le facem pentru a trece dintr-un șir de caractere în alt șir de caractere.

Putem compara conținutul extras de către program cu un conținut extras manual din pagină. Pentru a reduce câteva diferențe irelevante din text, spațiile și caracterele de final de rând nu vor fi luate în considerare. Pentru a nu defavoriza paginile cu mult conținut text, vom normaliza distanța Levenshtein prin împărțirea acesteia la numărul de caractere din textul extras manual. Valoarea normalizată ne va oferi și o idee despre cât de mult a greșit algoritmul(fie prin ștergerea conținutul relevant, dacă avem mai mult text extras de către algoritm decât manual, fie prin greșeală de a nu elimina conținut irelevant în caz contrar).

	Conţinut extras manual	Conţinut extras algoritmic	Distanţa Levenshtein	Distanţa împărţită la lungime
Articol HackerNews 1	5600	5792	924	0.165
Articol HackerNews 2	3272	3639	663	0.202
Articol HackerNews 3	3861	4283	608	0.157
Articol The Guardian Article 1	6107	6902	1586	0.259
Articol The Guardian Article 2	5410	5940	850	0.155

2.5 Căutare de conţinut după cuvinte cheie

Având de la pasul anterior și un șablon respectat de paginile cu conținut text, putem să ne folosim de acesta pentru a identifica alte pagini ce respectă același șablon și să vedem dacă cele mai importante cuvinte din textul din pagina respectivă se numără printre cuvintele căutate.

După găsirea șablonului, algoritmul continuă să navigheze în mod aleator către alte pagini legate de pagina curentă prin URLuri(aceeași idee ca și persoana care navighează aleator pe web, din lucrarea lui Larry Page).

Algoritmul este mult prea lung pentru a fi listat în lucrare, dar pașii pe care îi urmează sunt aceștia:

1. Se pleacă de la mulțimea de URLuri de pe pagina principală reunită cu mulțimea de URLuri de pe paginile folosite pentru obținerea șablonului. Din această mulțime se vor scoate URLuri cu speranța că vor conține text relevant și cuvintele cheie vor corespunde.
2. Căutarea aleatoare se va desfășura într-o buclă infinită ce se va opri doar dacă mulțimea de URLuri de la pasul 1 va fi goală..
3. Dacă o pagină corespunde șablonului(adică media deviațiilor standard la nivel de element HTML este mai mică decât valoarea de prag) atunci aceasta pagină conține text, dar înainte de a rula și restul algoritmilor(filtrarea și găsirea cuvintelor cheie) putem face o verificare superficială prin care ne vom uita dacă DOMul paginii conține măcar unul din cuvintele cheie(acest lucru nu este suficient deoarece nu știm încă dacă acel cuvânt este și important).
4. Dacă condițiile de pasul anterior sunt îndeplinite, putem aplica toate tehnicile de filtrare și putem extrage cuvintele cheie, pentru a verifica dacă acestea corespund cu ele căutate.

Câteva note importante ce nu ţin strict de algoritmul descris anterior:

1. Dacă a fost găsită o pagină ce respectă șablonul găsit, atunci toate URLurile de pe aceasta pagină sunt adăugate în lista de URLuri pe care algoritmul poate merge în continuare(dacă am dori să fim mai stricți, am putea adăuga URLuri la lista doar după ce este respectată și cea de-a doua condiție, pagina să conțină unul din cuvintele căutate).
2. Pentru a facilita căutări mai rapide în viitoare pentru alte cuvinte cheie, vom stoca pe disc o bază de date în format “JavaScript Object Notation”(JSON). Vom stoca datele sub forma unui dicționar unde cheia va fi URLul paginii iar valoarea va fi vectorul de cuvinte cheie. Pentru a nu îngreuna și mai mult algoritmul cu scrieri repetate pe disc la fiecare pagină, algoritmul va salva date noi doar periodic. La alte căutări viitoare, ne vom uita întâi în baza de date de tip JSON și apoi vom continua căutarea aleatoare.
3. Dacă vom găsi o pagină ce conține printre cele mai importante cuvinte și cuvintele căutate, atunci vom marca toate URLurile de pe această pagină drept prioritare în restul căutării(pentru a găsi restul de N-1 documente, un N este numărul de documente în care trebuie să găsim cuvintele căutate). URLuri prioritare vor fi puse la începutul mulțimii pentru fi procesate

primele, cu motivația că de obicei autorii de conținut grupează paginile cu tematica similară și adaugă, manual sau automat, o rubrica de pagini similare cu URL-uri către aceastea.

2.6 Sumarizarea conţinutului folosind Word2Vec şi PageRank

Profitând de segmentarea la nivel de propoziții și cuvinte făcută anterior putem calcula un indice de similaritate între propoziții, putând astfel să le introducem în algoritmul PageRank sunt forma unui graf ponderat, având ca ieșire propozițiile sortate după importanța lor. Librăria ce implementează algoritmul PageRank preia ca date de intrare o matrice de dimensiune NxN, unde N este numărul de propoziții din textul ce trebuie sumarizat, iar intersecția dintre o coloană și o linie reprezintă similaritatea propozițiilor de la indecșii respectivi.

```
1. string[] sentenceOne = scrapingResult.sentencesWords[sentenceOneIdx].ToArray();
   string[] sentenceTwo = scrapingResult.sentencesWords[sentenceTwoIdx].ToArray();

2. float[] sumSentenceOne = new float[databaseFeatureSize];
   Array.Clear(sumSentenceOne, 0, sumSentenceOne.Length);
   float[] sumSentenceTwo = new float[databaseFeatureSize];
   Array.Clear(sumSentenceTwo, 0, sumSentenceTwo.Length);

3. ComputeSentenceVector(sentenceOne, ref sumSentenceOne);
   ComputeSentenceVector(sentenceTwo, ref sumSentenceTwo);

4. float ratioOne = 1.0f / sumSentenceOne.Sum();
   sumSentenceOne = sumSentenceOne.Select(o => o * ratioOne).ToList().ToArray();
   float ratioTwo = 1.0f / sumSentenceTwo.Sum();
   sumSentenceTwo = sumSentenceTwo.Select(o => o * ratioTwo).ToList().ToArray();

5. documentMatrix[sentenceOneIdx].Add(new float());
   documentMatrix[sentenceOneIdx][sentenceTwoIdx] = Utils.CalculateCosineSimilarity(sumSentenceOne,
                                                                                    sumSentenceTwo);
```

Cum am descris în partea teoretică, ne vom folosi de segmentarea la nivel de cuvânt(linia 1) pentru a obține vectorii suma pentru fiecare cuvânt din propoziției, pe baza vectorilor la nivel de cuvânt din Word2Vec(liniile 2 și 3). Este util să normalizăm valorile din vectorii sumă, astfel încât suma elementelor dintr-un vector să fie egală cu 1(linia 4). Într-un final, vom completa celula din matrice cu indicele de similaritate dintre cele două propoziții, folosind similaritatea cosinus, adică produsul scalar dintre cei doi vectori împărțit la produsul lungimii vectorilor:

```
public static float CalculateCosineSimilarity(float[] vecA, float[] vecB)
{
    float dotProduct = vecA.DotProduct(vecB);
    float magnitudeOfA = (float)Math.Sqrt(vecA.DotProduct(vecA));
    float magnitudeOfB = (float)Math.Sqrt(vecB.DotProduct(vecB));

    return dotProduct / (magnitudeOfA * magnitudeOfB);
}
```

După ce vom introduce matricea de similaritate între propoziții în algoritmul PageRank, vom primi propozițiile sortate descrescător după importanța lor și vom putea alege un număr dintre acestea. Pentru acest scop am adăugat un parametru în interfața grafică ce va spune algoritmului câte cuvinte trebuie să aibă rezumatul conținutului, iar algoritmul va lua cele mai importante propoziții din vector și le va concatena la rezultatul final până când rezultatul final va fi mai lung decât parametrul ales(dar nu va trunchia ultima propoziție pentru că rezumatul să aibă exact lungimea cerută).
Deoarece este mai dificil să testăm algoritmic eficiența sumarizarii (nu avem o apreciere matematică precum distanța Levenshtein la pasul anterior), vom prezenta doar un caz de sumarizare al articolului [21] de pe HackerNews:

"Dubbed the newly spotted cyberattack campaign leveraged Microsoft file-sharing services including Sway, SharePoint launch highly targeted phishing attacks. According to a report Group-IB Threat Intelligence team published today and shared with The Hacker News, PerSwaysion operations attacked executives of more than 150 companies around the world, primarily with businesses in finance, law, and real estate sectors. Among these high-ranking officer victims, more than 20 Office365 accounts of executives, presidents, and managing directors appeared. So far successful and still ongoing, most PerSwaysion operations were orchestrated by scammers from Nigeria and South Africa who used a Vue.js JavaScript framework-based phishing kit, evidently, developed by and rented from Vietnamese speaking hackers. By late September campaign has adopted much mature technology stacks, using Google appspot for phishing web application servers and Cloudflare for data backend servers."

Propozițiile alese de către algoritm sunt din toate părțile articolului(prima propoziție din sumar corespunde cu prima propoziție din articol, iar ultimele propoziții sunt alese din partea de final a articolului).

2.7 Prezentarea interfeţei grafice şi a modului de utilizare

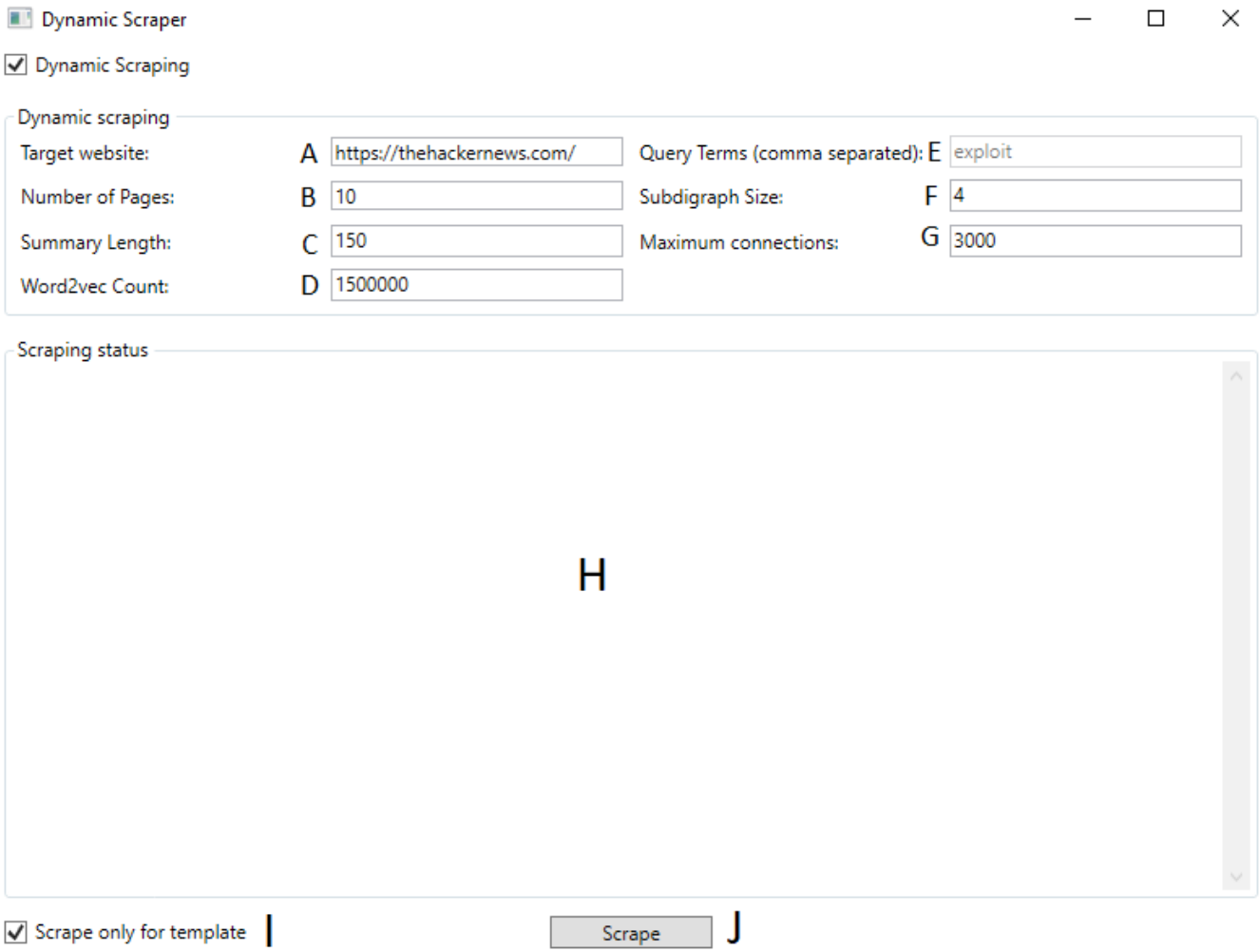

Figura 2.1 Intefața grafică

A) URLul către pagina de plecare de pe site, de obicei pagina principală.

B) În cazul căutării de pagini ce au ca şi cuvinte cheie cuvintele menţionate în caseta text F, vom alege un număr de pagini la care algoritmul să se oprească.

C) Lungimea rezumatului exprimată în numărul de cuvinte din acesta.

D) Numărul de cuvinte ce vor fi încărcate pentru modelul Word2Vec. Variabil în funcţie de posibilităţile de memorie RAM.

E) Cuvintele cheie după care se va uita algoritmul de căutare.

F) Mărimea subgrafului din care vom extrage şablonul. Patru este o valoare standard, folosită şi de autorii din lucrarea [4].

G) Numărul maxim de conexiuni din graf. Dacă vom trece de acest numar fără să găsim un subgraf ideal, vom alege cel mai bun subgraf de până la acest moment.

H) Deoarece execuţia programului este destul de lungă, vom printa în mod asincron mesaje către interfaţă pentru a informa utilizatorul la ce parte a procesului se află algoritmul.

I) Putem rula algoritmul într-un mod mai rapid în care să găsească doar şablonul folosind paginile din subgraf şi să aplice şi sumarizarea de acestea.

J) Butonul ce porneşte algoritmul cu parametrii setaţi în mod curent în interfaţă.

Concluzii

Domeniul extragerii de conținut este foarte bine dezvoltat și tehnologiile existente în prezent sunt foarte puternice. Soluția dezvoltată combină algoritmi din mai multe ramuri ale informaticii și matematicii: teoria grafurilor, învățare automată și procesarea limbajului natural. După cum am văzut, partea de extragere și filtrare a conținutului dă rezultate bune odată ce a fost găsit șablonul paginilor, iar partea de sumarizare a conținutului păstrează esențialul, profitând de faptul că sumarizarea se face extractiv unde sumarul reprezintă o selecție de propoziții din conținutul dat ca intrare.

Una dintre realizările avute pe perioada dezvoltării a fost eleganța cu care pot fi îmbinate concepte diferite pentru scopuri diferite. Partea de NLP din aplicație a fost menită pentru a facilita segmentarea textului pentru a putea alege cele mai importante cuvinte, dar aceeași segmentare a fost folosită și pentru filtrare prin eliminarea propozițiilor fără verbe sau pentru a facilita folosirea Word2Vec. De asemenea, Word2Vec a fost inițial folosit doar pentru sumarizarea conținutului, dar acesta s-a dovedit util și la filtrarea conținutului.

Un punct slab al metodei curente, admis și de autorii din lucrarea [4] este posibilitatea ca algoritmul să se piardă pe pagini cu foarte multe URLuri și să aleagă un subgraf greșit(în care paginile din subgraf chiar au același șablon, dar aceastea nu sunt pagini de conținut). De asemenea, algoritmul procesează foarte multe URLuri, iar acest pas este foarte încet datorită accesului la rețea și faptul că algoritmul nu poate rula alte sarcini în mod asincron până când cererea HTTP ajunge înapoi, din cauza necesității DOMului la fiecare pas.

Modificările aduse algoritmilor din lucrările [4] și [5] s-au dovedit benefice. Metoda de similaritate a structurii HTML pe baza de medie a deviațiilor standard s-a dovedit a fi foarte flexibilă datorită capacității de a fi parametrizabila, iar eliminarea pasului de fuziune a nodurilor HTML din [5] și adăugarea pașilor de filtrare pe bază de NLP și Word2Vec oferă o filtrare parametrizabilă și eficientă.

O posibilitate interesantă de continuare a lucrării ar fi împărțirea conținutului într-o bază de cunoștințe(prin metode mai simple de NLP sau prin învățare automată) pentru a putea răspunde cu limbaj natural la interogări asupra conținutului găsit(dacă vom rula algoritmul pentru a găsi sute de pagini spre anumite cuvinte cheie). O altă posibilitate ar fi adăugarea unui modul de analiză a sentimentalității cuvintelor din text pentru a putea sorta paginile găsite după sentimentul dominant. O astfel de încercare a fost făcută, algoritmul având o listă de cuvinte cu un anumit scor(cuvintele cu tentă pozitivă având un scor pozitiv și cele negative unul scor negativ), dar o astfel de abordare este susceptibilă la subiectivitate din cauza faptului că unele tipuri de articole vor avea în mod automat

cuvinte ce pot fi catalogate drept negative(de exemplu, știrile din securitatea cibernetică vor descrie în principal atacuri folosind cuvinte negative). O astfel de sortare a paginilor ar fi utilă în cazul în care vom căuta un număr foarte mare de pagini.

Tabel de figuri

Bibliografie

1. Erdinç Uzun, Alpay Doruk, H. Nusret Buluş, Erkan Özhan: Evaluation of HAP, AngleSharp and HTML document in web content extraction, International Scientific Conference, Gabrovo 18 Noiembrie 2017.
2. Soderland, S.: Learning information extraction rules for semi-structured and free text. Mach. Learn. 34(1-3), 233–272 (Februarie 1999).
3. Muslea, I., Minton, S., Knoblock, C.: Hierarchical wrapper induction for semistructured information sources. Auton Agents and Multi-Agent Syst pp. 1–28 (2001).
4. Julian Alarte, David Insa, Josep Silva, Salvador Tamarit: Web Template Extraction Based on Hyperlink Analysis. S. Escobar (Ed.): XIV Jornadas sobre Programación Y Lenguajes, PROLE 2014, Revised Selected Papers EPTCS 173, 2015, pp. 16–26.
5. Qingtang Liu, Mingbo Shao, Linjing Wu, Gang Zhao, and Guilin Fan: Main Content Extraction from Web Pages Based on Node Characteristics. Journal of Computing Science and Engineering, Vol. 11, No. 2, Iunie 2017, pp. 39-48.
6. Sergey Brin, Larry Page: The anatomy of a large-scale hypertextual Web search engine. 1998.
7. Rada Mihalcea, Paul Tarau: TextRank: Bringing Order Into Texts. Department of Computer Science. University of North Texas. 2004.
8. Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, Jeffrey Dean: Distributed Representations of Words and Phrases and their Composionality. Google 2013.
9. WWW Foundation: History https://webfoundation.org/about/vision/history-of-the-web/
10. Tim Berners-Lee (CERN), Daniel Connolly (Atrium): Hypertext Markup Language (HTML) - A Representation of Textual Information and MetaInformation for Retrieval and Interchange. Iunie 1993.
11. Apache OpenNLP Manual https://opennlp.apache.org/docs/1.9.2/manual/opennlp.html
12. SharpNLP https://archive.codeplex.com/?p=sharpnlp
13. Andrew McCallum, Dayne Freitag, Fernando Pereira: Maximum Entropy Markov Models for Information Extraction and Segmentation.
14. Adwait Ratnaparkhi: A Maximum Entropy Model for Part-Of-Speech Tagging.
15. Term frequency, Inverse Document Frequency: http://www.tfidf.com/.
16. Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *ICLR Workshop*, 2013.

17. Tomas Nikolov, Wen-tau Yih, Geoffrey Zweig: Linguistic Regularities in Continuous Space Word Representations.
18. Paritosh Pandey, Monica Wani: The C# Programming Language.
19. Windows Presentation Foundation - https://docs.microsoft.com/en-us/dotnet/framework/wpf/
20. XAML Overview – https://docs.microsoft.com/en-us/dotnet/desktop-wpf/fundamentals/xaml .
21. https://thehackernews.com/2020/04/targeted-phishing-attacks-successfully.html