Tartalomkinyerés futásidőben generált weblapokról

Komporday András

Konzulens:

Indig Balázs,

Dr. Prószéky Gábor

Mérnök Informatikus BSc

szakdolgozat

Pázmány Péter Katolikus Egyetem

Információs Technológiai és Bionikai Kar

2016. május 7.

Alulírott Komporday András, a Pázmány Péter Katolikus Egyetem Információs Technológiai és Bionikai Karának hallgatója kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, és a szakdolgozatban csak a megadott forrásokat használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen a forrás megadásával megjelöltem. Ezt a Szakdolgozatot más szakon még nem nyújtottam be.

Tartalomjegyzék

Kivonat

Az internet terjedésével egyre fontosabbá válik az a törekvés, hogy az ott fellelhető információ ne csak a felhasználók, hanem különböző szoftverek által is feldolgozható legyen. Ebből a célból jött létre a 2000-es évek elején a szemantikus web koncepciója, majd emiatt születtek meg a különböző tartalomkinyerési technológiák.

A különböző célokra specializált tartalomkinyerő programok célja az, hogy a feladatuk szempontjából releváns információkat a lényegtelen információktól (ún. boilerplate) megtisztítva bányásszák ki a weboldalakból. Ilyen feladat lehet pl. a nyelvészeti célú felhasználás, illetve az információ kis méretű eszközökön jobban megjeleníthetővé tétele és a látássérültek számára készült képernyőolvasó programok által könnyebben feldolgozható formátumba történő alakítása. Segítségükre vannak feladatuk végrehajtásában az ún. keresőrobotok vagy crawlerek, a weblapokat felderítő és letöltő szoftverek, melyek folyamatosan új és új oldalakkal látják el a kinyerőalgoritmusokat.

A modern tartalomkinyerő rendszerek és a velük összhangban működő keresőrobotok egyik legnagyobb kihívása, hogy képesek legyenek kibányászni az internet egyre nagyobb hányadát elfoglaló dinamikusan, futási időben generált weblapok releváns tartalmait. Ebből a szempontból a web rohamtempóban fejlődve, előbb a PHP-alapú, kliensoldalon generált weblapokkal, majd a többnyire Javascriptre támaszkodó, kliensoldali aktivitásra változó felületekkel maga mögött hagyta a webbányászatot, mivel annak csúcstechnológiát jelentő képviselői sem képesek ezen adatok teljes mértékű feldolgozására.

Jelen dolgozat célja, hogy röviden bemutassa a dinamikusan generált weblapok működésének alapjait, ismertesse a tartalomkinyerés és ezzel együtt a weboldalak felderítése, a crawling kihívásait ezen a területen és prezentáljon egy lehetséges megoldást[1] *a probléma kezelésére. Ilyen módon szűkíthető lenne a tartalomgenerálás- és kinyerés között utóbbi évtizedben kialakult technológiai rés.*

[1] `https://github.com/kiskompi/RTExtractor`

Abstract

With the continuous growth of the web the intention to process its data via automatized softwares became increasingly important. Due to this, in the early 2000s the first idea of the semantic web and later the automatized crawling and data extraction were born.

The data extraction technologies, many of them created with different purposes, have a common goal - extract all of the data which is relevant from their point of view. This can be Natural Language Processing, commercial data analytics, or even end-user usage, eg. Increasing the readability on small-screen devices or for the visually impaired. These extractors usually work together with so-called crawlers, small softwares designed to explore and download websites.

One of the greatest challenges of these data extraction tools and crawlers is to effectively mine the server-side generated, PHP-based and the modern Javascript-heavy, runtime and client-side generated web pages. In this matter the web evolved rapidly and left behind the classic data extraction technologies. Due to this technological gap, even the state-of-the-art technologies can not process effectively these websites.

The goal of this paper is to quickly introduce the dynamic web technologies and data extraction, describe the challenges of the crawling and extraction on the dynamic web sites and finally propose an idea[2] to solve this problem. This way, the gap which formed between the web and data extraction in the last decade could be narrowed.

[2] `https://github.com/kiskompi/RTExtractor`

I. rész

Bevezetés

Napjainkra az internet a személyes tájékozódás és a tudományos adatgyűjtés számára is az egyik legfontosabb információforrássá vált. Az óriási adatmennyiség azonban új kihívások elé állította a kutatókat, megszületett az adatbányászat: a keresőrobotok és a tartalomkinyerő algoritmusok képessé váltak automatizáltan felkutatni a releváns adatokat és kinyerni azokat a weblapok forráskódjából, az ún. *boilerplate* tartalmak közül. Ám a szöveg kinyerés és feldolgozás technikája az utóbbi időben lemaradt a futásidőben generált oldalak megjelenésével. A következő fejezetekben az alapoktól kezdődően tekintem át a jelenlegi helyzetet és javasolok egy lehetséges megoldást.

1. Alapfogalmak

1.1. A boilerplate

A *boilerplate* az összefoglaló elnevezése azon dolgoknak, amelyek egy dokumentumban egy feladat végrehajtása szempontjából lényegtelenek és általában több dokumentumban változatlan, vagy alig változó formában jelennek meg. Eredetileg ilyenek a jogi dokumentumokban gyakran használt visszatérő kifejezések és dokumentumról dokumentumra ismétlődő szövegrészek. A mi esetünkben ezek megfeleltethetők a különböző weboldalakon gyakran szereplő a menüknek, lábléceknek és fejléceknek ahogy az A.1 és az A.2 függelékekben látható.

Mivel a különböző feladatok számára különböző tartalmak jelenthetnek releváns információt, így míg pl. egy nyelvészeti célú szoftvernél célszerű kiszűrni a navigációs menüket, addig egy látássérültek számára tervezett szűrőnek meg kell azokat hagynia, hogy a felhasználó továbbra is rendeltetésszerűen tudja használni az oldalt. Ezek miatt az eltérések miatt a boilerplate-re nem lehet egy általános definíciót adni. Ehelyett különböző szervezetek különböző irányelveket dolgoztak ki a különböző feladatokra specializált szoftverek

tervezésének szabványosítására. Az alábbiakban a *CleanEval*[3] és a *KrdWrd*[4] nyelvészeti célú irányelv-gyűjteményeket mutatom be.

A **cleanEval** irányelvek a következő elemeket veszi bele a boilerplate fogalmába [1]

- Navigációs elemek
- Linklisták
- Szerzői jogi nyilatkozatok
- Sablonelemek, pl. fejlécek és láblécek
- Hirdetések
- Web-spam, pl. spammerek automatikusan létrehozott bejegyzései
- Formok
- Másolt tartalmak, pl. egy előző hozzászóló idézése egy fórumban.

1.2. A tartalomkinyerés

A tartalomkinyerés fő céljai a weboldalak boilerplate tartalmainak szűrése, hogy később a kinyert lényeges tartalmat különböző feladatokban fel lehessen használni. Ilyen például, hogy könnyebben olvasható formára hozza az oldalt kisméretű eszközökön, történő fogyasztás, illetve látássérültek számára készült képernyőolvasó programok (pl. JAWS) – segítségével jobb feldolgozhatóság céljából.

Az előbbi feladatkör végrehajtására két fő módszer lett kidolgozva:

- a *content reformatting*, azaz a tartalom újraformázásán alapuló tartalom kinyerési módszerek, melyek alkalmazása esetében az oldal a tartalom közvetlen módosításával, a lényegtelen információt formázással a képernyőn kívülre helyezve, a képeket és linkeket elrejtve, és a betűméretet megnövelve nyeri el a végső formáját.

[3] `http://cleaneval.sigwac.org.uk`
[4] `https://github.com/krdwrd/`

- Továbbá léteznek a *content extraction* (továbbiakban: CE), azaz olyan tartalom kinyerési technikák, melyek használatakor a weboldal lényeges tartalma megmarad, de bolierplate elemeket a tartalomkinyerő eldobja.

Jelen dolgozat célja az utóbbi elven működő technológiák összehasonlítása. A *content extraction* módszereknél, a *content reformatting*gal ellentétben a lényegtelen részek a HTML struktúrából is eltűnnek, ami egyrészt csökkenti az adatfelhasználást, másrészt a HTML parserek és képernyőolvasók így pontosabban tudnak működni, ezek ugyanis nem a képernyőre renderelt információ, hanem a HTML kód alapján dolgoznak.

A CE módszerek működésének legfontosabb kritériumai a valós idejű *(real time)* működés, az adatforgalom csökkentése és az, hogy használhatók legyenek bármilyen felépítéssel rendelkező honlapokon: nem tekinthető ideálisnak egy olyan módszer, mely egy típusú weblapon működik, azonban az algoritmus bedrótozott *(hardcoded)* tulajdonságai miatt attól különböző felépítésű honlapokon működésképtelen.

2. Az összehasonlítás módszertana

A tartalomkinyerési módszerek elemzésének és összehasonlításának legfőbb szempontjai

- az on-the-fly működési sebesség (adatmennyiség függvényében),
- a stabilitás, azaz az ugyanazon a honlapon elért eredmények hasonlósága többszöri futtatás során,
- illetve a különböző felépítésű honlapokon való működés pontossága (nem hagyhat bent sem túl sok, sem túl kevés tartalmat).

Emellett fontos megnézni, hogy az adott technológia alkalmazható-e többféle nyelven íródott webhelyeken, illetve, hogy felhasználható-e többféle célra (mint felhasználói vagy nyelvészeti célú alkalmazás).

3. A webes környezet elemzése tartalom kinyerési szempontból

Ebben a fejezetben a legnépszerűbb magyar- és nemzetközi honlapokat elemezzük a webes tartalomkinyerést befolyásoló tulajdonságai alapján. Az összehasonlítás legfontosabb szempontjai

- a javascript használata,
- az @media tagek használata a CSS-ben,
- a HTML dokumentumok szabványkövetése
- és a paraméterérzékenység.

A weblapok kiválasztásában szempontként megjelent a látogatottságuk, funkciójuk, illetve az, hogy eltérő felépítésű és technikai hátterű felületek is részt vegyenek a tesztekben. Ezzel kívántuk modellezni a legtöbb nagy forgalmú weboldal dolgozat írása kori felépítését. A vizsgált weblapok fő csoportját a híroldalak, illetve hagyományos elrendezésű honlapok (egy hosszabb szöveges törzs *(main content)*, mellette rövidebb szöveges információval rendelkező keret *(noisy content, boilerplate)* tartalmúak) tették ki. A másik fő csoport a blogok (ezekben a fő tartalom kevésbé hosszú és hangsúlyos), illetve a fórumok voltak. Utóbbiak több szempontból is speciálisak, mivel a releváns tartalom (a hozzászólás) egyrészt rövid szöveges rész, másrészt magas *link-szöveg aránnyal* (link-to-text ratio) rendelkezik. A honlapok elemzésekor minden esetben külön vettem a címlapot és a cikkeket tartalmazó aloldalakat, mivel, bár az utóbbiak könnyen szűrhető, többnyire hosszú szöveges fontos tartalommal rendelkeznek a zajos tartalom mellett, addig az előbbiek, hasonlóan a webshopokhoz és fórumokhoz, magas link-szöveg arányú, rövid szövegeket tartalmazó egységekből állnak.

Az webes környezet elemzése során a legnagyobb hazai honlapokból válogattam. A válogatás közben figyelembe vettem az oldalak látogatottságát, profilját, felépítését és technikai hátterét (mennyire korszerű megoldásokat alkalmaznak?), illetve a teljesség kedvéért egy nagy külföldi oldalt is kiválasztottam. A tesztben szereplő webhelyeket igyekeztem úgy kiválasztani, hogy mindegyik egy-egy nagyobb csoportot reprezentáljon annak tulajdonságai alapján. A tesztben szereplő honlapok listája:

- Az **index.hu** reprezentálja a modern technikai háttérrel működő, erősen dinamikus, de többnyire hosszabb szövegekkel operáló hírlapokat. A főoldalon a képes tartalom dominál, a belső linkeket a *dex.hu* domainen keresztül irányítják át. Ebből a fajtából az elterjedtsége miatt megnéztem még külföldi példának a **bbc.com**-ot is.

- Az **origo.hu** a szintén erősen dinamikus, de korábbi szabványokkal dolgozó (régebbi), több képpel, gyengébb technikai minőségű oldalakat képviseli.

- Az **euronews.com** egy többnyelvű weboldal, melyben lenyíló listával lehet kiválasztani, hogy a cikkek milyen nyelven jelenjenek meg.

- A **port.hu** a sok képpel, multimédiás tartalommal, viszonylag rövid szövegekkel és magas link-szöveg–aránnyal rendelkező dinamikus lap.

- Az **jofogas.hu** a webshopok bemutatására, melyek jellemzően magas link-szöveg aránnyal, képekkel és kevés és rövid szöveggel rendelkeznek.

- Az **idokep.hu** weboldal szinte nulla szöveges tartalommal, kizárólag képekkel és szkriptekkel dolgozik.

3.1. Akadálymentes honlapok

Az internet robbanásszerű fejlődésével, a képes és multimédiás tartalmak, szkriptek elterjedésével a weblapok felhasználói élménye egyre jobb és jobb lett. Ezzel párhuzamosan azonban nőtt a zajos, nem releváns tartalom tartalom mennyisége az oldalakon - jelenleg ez hozzávetőleg átlagosan 40-50% [2]. A design és képi elemek egyre hangsúlyosabbá válása ugyanakkor egyre nehezebbé teszi az internet használatát a vak és gyengénlátó, képernyőolvasó programokat használó felhasználóknak. A W3C 2008-ban dolgozta ki a WCAG 2.0 (ISO 40500)[5] szabványt, mely leírja az akadálymentes weblapok készítésének alapelveit [2].

Az akadálymentes weblapok és a tartalomkinyerés közötti szoros kapcsolatra több tanulmány és szerző is felhívta már a figyelmet (pl. Suhit Gupta [3] és Emilio Ferrara [4]). A

[5] `https://www.w3.org/TR/WCAG20/`

tartalomkinyerés és az akadálymentesség kapcsolata kétirányú. Míg a WCAG 2.0 web-akadálymentesítési szabvány figyelembevételével készült weblapok tartalmának kinyerése pontosabb és könnyebb a kevesebb zajos (boilerplate) tartalom miatt, addig a kevésbé akadálymentes, nem szabványkövető weblapok használatát a tartalomkinyerő szoftverek tudják megkönnyíteni a fogyatékkal élő felhasználók számára, a boilerplate tartalom kiszűrésével. A két téma közötti kapcsolat kétirányú: egyrészt az akadálymentes webhelyek kialakításának alapvető lépése pl. az *img* tagek *alt* attribútumának következetes használata (a képernyőolvasó programok ezt az attribútumot használják a kép tartalmának kiolvasására), illetve a túlzott Javascript használat visszaszorítása. A szabvány szerint elkészített honlapok nem csak a fogyatékossággal élők számára biztosítanak könnyebb navigációt, hanem pl. az alacsonyabb sebességű interneteléréssel rendelkező felhasználók számára is könnyebben elérhetővé teszik a tartalmat, illetve jobb vizuális élményt nyújtanak, átláthatóbb felépítéssel rendelkeznek. A Crunch projectnek egyik megfogalmazott célja volt a képernyőolvasó programokat használók böngészésének megkönnyítése és a képernyőolvasókkal való minél ideálisabb együttműködés [3].

3.2. Javascript használata

A modern weblapok nagy része erősen épít a Javascript használatára. Bár tényleg esztétikus megoldásokat lehet elérni ennek használatával, helytelen vagy túlzott használatával elérhetetlenné, vagy nehezen elérhetővé válhatnak a weblapok egyes részei a képernyőolvasó programokat használóknak, illetve nehézzé tehetik a tartalom kinyerését bizonyos algoritmusok számára. Ennek fő oka, hogy a Javascript szkriptek által dinamikusan változtatott és generált elrendezés túlságosan összeköti a webhely tartalmát annak megjelenítésével (pl. a display paraméter hidden-re állítása helyett a kódrészlet törlése a HTML-ből), így korlátozva a képernyőolvasó programok hatásosságát. Az WCAG 2.0 szabvány ajánlása az, hogy a weblapok dizájnját a CSS stíluslap (stylesheet) határozza meg, hogy teljesen elkülönüljön a tartalom a megjelenéstől. A tesztalanyként szolgáló weblapok vizsgálatakor a következőre jutottam:

Az összes vizsgált honlap nagy mennyiségű Javascriptet használ, ám egyedül az origo.hu-

nál váltak elérhetetlenné a lap egyes (ráadásul fontos) részei a szkriptek letiltásakor. Az index.hu-n csak bizonyos dinamikusan generált elemekbe beágyazott képek tűntek el. A bbc.com esetén a legtöbb beágyazott kép és videó eltűnt, de szöveges tartalom nem vált elérhetetlenné. A port.hu-n nem történt változás a Javascript letiltását követően.

3.3. Az @media tagek használata

Az @media tagek a CSS stíluslapban használatosak a különböző típusú megjelenítőkhöz illeszkedő stílusparaméterek dinamikus kialakításában. Ilyen @media *media query*-kel (CSS 3.0 előtt *media types*) dolgozó weblapok ideálisabban jelennek meg különböző eszközökön, pl. kisméretű mobileszközökön és nyomtatókon. Az @media tagek le tudják kezelni a megjelenítő eszköz méreteit, a *viewport* méreteit, a képernyő tájolását (álló vagy fekvő?) a felbontást és egyebeket.

A jelenlegi CSS verzióban (CSS3) a következő típusú kimeneti perifériákat (*output peripherials*) különbözteti meg: nyomtató (*@media print*), elektronikus képernyők (monitorok, tabletek, telefonok – *@media screen*) és képernyőolvasó programok (*@media speech*). Az algoritmusok figyelembe vehetik azt, hogy a print és speech @media tageket használó megjelenítési módok alapján könnyebb kinyerni a releváns információt, mivel azok kevésbé zajosan, kevesebb kiegészítő tartalommal látják el azt.

Az egyes honlapok forráskódjának elemzésekor arra lettem figyelmes, hogy legtöbb helyen az @media tagek közül csak a *print*et használják, a *speech* taget nem ismerik, illetve a *print* tag használatakor is több helyen csak letiltották a CSS használatát, érdemi alakítást a honlapon nem végeztek. Az index.hu vizsgálatakor azt láttam, hogy kisméretű eszközökön csak áthelyezte, nyomtatási nézetben viszont teljesen kiszűrte az irreleváns tartalmakat. Az origo.hu ezzel ellentétben nem rendelkezik nyomtatási nézettel, ott a felhasználók hozzászólásaival együtt, változatlan formában jelenik meg a felület. A BBC online felületén (bbc.com) a nyomtatási nézet megegyezik az álló tájolású képernyős nézettel, ahol a zajos tartalom csak áthelyeződik, de nem tűnik el. A port.hu-nak nincs nyomtatási nézete, a kinyomtatott oldalak változatlanul jelennek meg a nyomtatott papíron is.

A weboldalak más csoportját képezik a webshopok, melyek erősen dinamikus, gyakran

sok szkriptet futtató oldalak, ahol a releváns információ sokszor nem tartalmaz nagy mennyiségű szöveget, így az ilyen elemzést használó algoritmusok (pl. JusText) nem használhatók esetükben. A webshopokhoz hasonlóan dinamikus és kevés szöveggel operáló oldalak az időjárás előrejelzésével foglalkozó oldalak (pl. idokep.hu). Ezek sokszor nagyon sok képet és kevés szöveges információt tartalmaznak, továbbá sokszor összetett és nagy mennyiségű szkripteket is.

3.4. A HTML-dokumentumok szabványkövetése

A tartalomkinyerő algoritmusok és XML parserek alkalmazását jelentősen megkönnyíti, ha az adott weblap betartja a HTML aktuális verziójának (HTML 5[6]) szabvány előírásait (a modern honlapok a legutóbbi 2001-es XHTML 1.0[7] szabványt annak elavulása miatt már nem tudják érdemben alkalmazni). A kevésbé szabványos honlapok a parserek számára kevésbé értelmezhetőek, elemzésük zajosabb eredményhez vezethet. A nem szabványos weblapokat az algoritmusok implementációjában szintén tudni kell kezelni, azaz az algoritmusoknak rendelkezniük kell egy „hibatűréssel". Mivel a legtöbb modern XML parser rendelkezik beépített hibajavító algoritmussal[8], így a szabványsértő tag-eket a tartalomkinyerő technológiáknak nem kell kezelnie. A weboldalak szabványosságát a html5.validator.nu html validatorral ellenőriztem.

A HTML5 szabványok betartásának ellenőrzésekor a következő eredmények születtek: az index.hu rendszeresen szabálytalanul használta a *header* tag-eket, illetve attribútumokat (pl. sok helyen hiányzott a képek *alt* attribútuma), nem követték a szabványt a HTML elemek egymásba ágyazásánál (pl. header tag-be a szabvány előírásai ellenére *div* tag-et tettek). A bbc.com szabálytalanul haszálta a *rev* és *rel* attribútumokat, illetve több *<a>* tag-et (a *href* értéke # volt). Az origo.hu szabálytalanul használt *name* attribútumot *div* tag-eknél, *pre-src* attribútumot *img* tag-eknél, illetve több elgépelést is tartalmazott a dinamikusan generált kód (ez utóbbiakat a HTML parserek könnyen javítják). A port.hu a

[6] `https://www.w3.org/TR/html5/`

[7] `https://www.w3.org/TR/xhtml1/`

[8] A *HTML tag soup* olyan HTML-kód, mely nem követi a szabvány előírásait, például rossz sorrendben lezárt HTML tag-eket tartalmaz. A tag soup-ok javítására több bevált módszer is született, pl. a HTML Tidy: `http://www.html-tidy.org`

meta tag-eket szabálytalanul kezelte, ám ez irreleváns, mivel a CE-k először mindig kiszűrik ezeket. Emellett CSS helyett tag attribútumokkal formázta a HTML elemeket, ami a HTML4 szabvánnyal ellentétes. HTML5 elemeket használ HTML4 típusú dokumentumban, elhagyja az *alt* attribútumot a képekről.

Összességében elmondható, hogy a legnagyobb honlapok sokszor nem használnak olyan attribútumokat, melyek a tartalomkinyerést és a képernyő felolvasó programok működését segítenék, pl. a képeknél az *alt*, szkripteknél *type* (bár ez kevésbé gyakori). A HTML dokumentumok szemantikai hibákat kevéssé tartalmaznak, néhány esetben (port.hu-nál elsősorban) azonban ez is előfordult. A CE algoritmusokat ennek megfelelően úgy kell kialakítani, hogy az XML parserek hibajavításán túl is legyen egy ún. *threshold*ja, azaz hibatűrése, és kevésbé szabványkövető oldalakat is kezelni tudjon.

4. További lehetőségek a tartalom kinyerésére és osztályozására

4.1. Hibaoldalak

A HTTP hibaoldalak olyan általános hibaüzeneteket megjelenítő oldalak, melyek akkor jelennek meg, ha a szerver által küldött HTTP üzenet fejlécében bizonyos meghatározott állapotkód szerepel. Az állapotkódokat az IETF RFC 7231-es szabványa definiálja [5]. Ilyen állapotkódok jelzik, ha az adott oldal nem található (404), ha a felhasználó jogosulatlan az oldal elérésére (401), stb. A HTTP állapotkódok öt osztályba sorolhatók, melyeket az állapotkód első számjegyével különböztetünk meg:

1 információs üzenetek, jelenleg nem használják, a HTTP 1.0 szabvány nem definiálja.

2 siker – az ebbe az osztályba tartozó üzenetek azt jelzik, hogy a kliens kérése megérkezett, fel lett dolgozva, el lett elfogadva és végre lett hajtva.

3 átirányítás – ezek az üzenetek azt jelentik, hogy a kliensnek további műveleteket kell végrehajtania a kérés végrehajtásához.

4 ügyfélhiba – a kliensek kérése nem tudott végrehajtódni, mivel a kérés hibás volt.

5 szerverhiba – a kérés érvényes volt, de a végrehajtás a szerver hibája miatt sikertelen.

A HTTP üzenetek fejléce egyszerűen leolvasható a proxyként működő tartalomkinyerő rendszerekkel, így a kinyert, egész biztosan releváns tartalom nélküli oldal elemeit nagy valószínűséggel boilerplate-nek jelölhetjük. Ezt a megközelítést kizárólag webhely-szintű, vagy tanuló algoritmusokkal operáló technológiákban lehet használni. Eddigi kutatásaim során nem találtam olyan technológiát, ami kihasználta volna az ebben rejlő lehetőségeket, így ennek implementálása ígéretes lehetőséget jelenthet a jelenlegi megoldások továbbfejlesztésében vagy egy új algoritmus létrehozásában.

4.2. Tartalomkezelő rendszerek

A dinamikus weblapok nagy része ún. *tartalomkezelő rendszerekre (Content Management System, továbbiakban CMS)* alapul. Ezek a CMS-ek biztosítják az oldal szerveroldali működését, a dinamikus oldalak generálását. Felületet biztosítanak az jogosultsággal rendelkező felhasználóknak az oldal működtetéséhez, fejlesztéséhez és az oldal tartalmának manipulálásához. A legnépszerűbb tartalomkezelő rendszerek a dolgozat megírásakor:

- A Wordpress rendszert használta 2015-ben a weboldalak 26,1%-a
- Joomla futott a weboldalak 2,7%-a alatt
- A Drupal pedig a lapok 2,2%-át működtette

Összesen az oldalak 44,2%-a használt valamilyen tartalomkezelő rendszert [6]. Az egyes CMS-ekkel generált oldalak többnyire a CMS-re jellemző felépítéssel rendelkeznek[9], és több megoldás is született arra vonatkozólag, hogyan lehetne meghatározni az oldalt működtető rendszert csak a HTML dokumentum ismeretében. Ha egy ilyen megoldást sikerülne beépíteni a tartalomkinyerő rendszerekbe, az azt jelentené, hogy a CMS-ek által dinamikusan generált oldalakról a felépítésük ismeretében sokkal könnyebben és pontosabban lehetne kinyerni a fontos tartalmakat.

[9]Web Template System: egy olyan informatikai rendszer, mely előre definiált sablonokból és egyedi tartalmakból, egy ún. sablon motorral (*tamplate engine*) generál nagy mennyiségű, azonos felépítésű weblapot. Ilyen technológiát használnak a tartalomkezelő rendszerek, webes alkalmazás-keretrendszerek és HTML-szerkesztők.

4.3. RSS feed

Az *RSS 2.0 (Really Simple Syndication)* egy rendszeresen változó tartalmú weblapok követésére létrejött XML formátumcsalád. Több híroldal, blog és más online felület használ ún. RSS Feedet, hogy a tartalmait megossza az arra feliratkozó felhasználókkal. Az RSS 2.0 szabvány egy könnyen áttekinthető XML formátummal rendelkezik, mely tartalmazza az RSS csatorna címét és leírását, az oldal URL-jét, majd a csatornában megjelenő cikkek címét, leadjét és egy linket a teljes cikkre. Egy egyszerű példa egy RSS kódra:

```
<?xml version="1.0"?>
<rss version="2.0">
  <channel>
    <title>Példa csatorna</title>
    <link>http://pelda.hu/rovat</link>
    <description>Az egyik rovat RSS csatornája</description>
    <item>
       <title>Egy cikk a rovatból</title>
       <link>http://pelda.hu/rovat/egycikk</link>
       <description>
         A rovat RSS-csatornájába felkerült cikk leadje.
        </description>
    </item>
    <item>
       <title>Egy másik cikk, lead nélkül</title>
       <link>http://pelda.hu/rovat/masikcikk</link>
    </item>
  </channel>
</rss>
```

Az RSS funkcionalitása szükség esetén XML modulokkal *(namespace)* bővíthető. Így lehet bejegyzéseknek egyedi azonosítót megadni (pl. DOI, ISBN, stb.). Bár az RSS széleskörűen

használttá vált, soha nem lett hivatalos szabvány. A technológia hibái és korlátjai miatt egyre többen gondolták úgy, hogy egy nemzetközi szabvány által szabályozott, kiszámíthatóan fejlesztett technológiára van szükség, ami végül az Atom szabványban teljesedett ki.

Az Atom egy 2005-ben, az IETF által kidolgozott szabvány, mely az RSS technológiát hivatott kiváltani. Az RSS-hez hasonlóan itt is egy XML-formátumról van szó, amely azonban több lehetőséget, rugalmasabb felhasználást és egy nemzetközi szabvány kiszámíthatóságát kínálja [7]. Egy egyszerű Atom kód:

```
<?xml version="1.0" encoding="utf-8"?>
<feed xmlns="http://pelda.hu/rovat'’>
  <title>Példa rovat</title>
  <subtitle>Ez egy példa oldal egy rovatána Atom Feed-je.</subtitle>
  <link href="http://pelda.hu/rovat/feed/" rel="self" />
  <link href="http://pelda.hu/" />
  <id>egy-egyedi-azonosito</id>
  <updated>2016-03-15T12:10:09Z</updated>
  <entry>
    <title>Az egyik bejegyzés címe</title>
    <link href="http://pelda.hu/rovat/egyikbejegyzes" />
    <link rel="alternate" type="text/html"
     href="http://pelda.hu/rovat/egyikbejegyzes.html" />
    <link rel="edit"
     href="http://pelda.hu/rovat/egyikbejegyzes/szerkesztes" />
    <id>bejegyzes-egyedi-azonositoja</id>
    <updated>2016-03-13T18:10:52Z</updated>
    <summary>Ez a cikk leadje.</summary>
    <author>
      <name>Major Anna</name>
      <email>majoranna@pelda.hu</email>
    </author>
```

```
  </entry>
</feed>
```

Mind az RSS, mind az Atom feldolgozásához ún. *feed readert*, vagy *aggregátort* kell használni. A legtöbb mai szoftver mindkét típust képes kezelni, ám a népszerűsége miatt az RSS sok helyen még mindig preferált formátum maradt.

Az RSS és Atom csatornák megkönnyíthetik a tartalomkinyerő algoritmusok dolgát is, hiszen az RSS csatornában szereplő zajmentes adatok alapján könnyebben felismerhetővé válhat az oldal releváns információtartalma. A tartalomkinyerésben történő használatuk legfőbb akadálya, hogy viszonylag kevés helyen alkalmazzák ezeket a technológiákat.

4.4. Belső linkek

A HTML dokumentumokban a más dokumentumokra és a dokumentum más részeire az <a> HTML taggel lehet hivatkozni. A hivatkozás helyét a tag *href* attribútumával lehet megadni, mely mutathat egy másik HTML dokumentumra, az aktuális dokumentum egy szektorára, illetve egy Javascript elemre. A különféle típusú linkek eltérő kezelése az adatkinyerés pontosságát növelheti, mivel így meg lehet különböztetni a weboldal más részeire mutató linkeket, a bannereket, az oldalon belüli hivatkozásokat és szkripthívásokat. A belső linkeket a súlyozással működő CE módszerek használatakor a külsőknél nagyobb súllyal lehet figyelembe venni, mivel nagyobb valószínűséggel jelentenek releváns tartalmat, mint a külső felületekre mutatók, melyek sokszor hirdetések, közösségi média-oldalakra és irreleváns helyekre hivatkoznak.

A belső linkeknek egy oldalon három fő feladatuk van:

- Biztosítják a weboldalon történő tájékozódást.
- Kialakítják az oldal hierarchiáját.
- Segítik a keresőoptimalizálást.

Sok weboldal a belső linkekre egy proxyn keresztül hivatkozik (pl. index.hu a dex.hu-n keresztül) a belső linkekre, mely proxyk az URL-ben megkapott paraméterek alapján irányítják át a felhasználót az elérni kívánt oldalra. Az ilyen típusú linkek megkülönböztetése

a külső oldalakra mutató linkektől nehezebb, mint az eredeti domainre, vagy relatív elérési útra mutató linkek kinyerése, ám ugyanúgy belső hivatkozásnak számítanak. Bár az internetes oldalak bejárásának, és így a belső linkek felderítésének problémája nem közvetlenül a tartalomkinyerés, inkább a crawlerek témakörébe tartozik, azonban webhelyszintű algoritmusok esetén a két problémakör annyira összefügg, hogy az előbbi tárgyalásakor az utóbbi témát nem lehet említés nélkül hagyni.

4.5. A szemantikus web

A szemantikus web egy eredetileg T. Berners Lee által a Scientific Americanben 2001-ben megálmodott XML és RDF alapú leírónyelv, melynek célja, hogy egy olyan, elsősorban parserek és szoftverek által feldolgozott hálózatot hozzon létre, melyen megvalósul az adatok egységes definiálása, jellemzése és értelmezése, a hagyományos internet humán felhasználók számára történt kialakítása helyett. Bár a szemantikus web, és így a tiszta világhálós információmegosztás csak tervszinten valósult meg, a tartalomkinyerő algoritmusok elmozdíthatják a hagyományos webet egy olyan irányba, amely a számítógépes programok számára könnyebben feldolgozható formátumban közzé tett információ segítségével megsokszorozza a crawlerek és kereső hatékonyságát és pontosságát. Megállapítható, hogy az eredeti, RDF-alapú, könnyen feldolgozható adathálózat kialakulása helyett egy olyan világháló-struktúra vált dominánssá, melyben az adatok emberi fogyasztásra szánt formában jelennek meg, és ezen adatokat egyre kifinomultabb tartalomkinyerő algoritmusok alakítják más, szoftverek által feldolgozható formátumba.

4.6. Futásidőben generált weblapok

A modern weboldalak egy része (pl. a közösségi média) nem csak oldalfrissítéskor, hanem a lap látogatási ideje alatt folyamatosan frissülnek, az újabb és újabb információkat közvetítve valós időben a felhasználó felé. Ezek az oldalak sokszor óriási adatmennyiséget kezelnek (a Facebook napi adatforgalma 2014-ben nagyjából 500Tb volt [8]). Ilyen mennyiségű adat feldolgozása egyrészt óriási lépés lehet a nyelvtechnológusok számára, másrészt olyan technikai kihívás, melynek megoldása a tartalomkinyerésben ma még többnyire kihasználatlan

technológiák és paradigmák alkalmazását követeli meg (pl. aszinkron és párhuzamos számítások).

Ezek a Javascript-től erősen függő oldalak sokszor nem szabványos HTML kódokat generálnak, így feldolgozásukkor figyelembe kell venni az alkalmazott technológiák hibatűrését.

4.7. HTML-elemzők (parserek)

A HTML oldalak feldolgozására és elemzésére készített technológiákat ún. HTML parsereknek nevezzük. Ezek a programok vagy programkönyvtárak képesek HTML oldalak vagy töredékek értelmezésére, különböző lehetőségeket (vizualizáció, elem listázás, DOM-fa bejárás vagy akár az oldal szerkezetének manipulációja) biztosítva a felhasználóknak. Weblapok feldolgozásakor gyakorlatilag elengedhetetlen az ilyen technológiák használata, és a megfelelő parser kiválasztása jelentősen megkönnyítheti a további munkafolyamatot.

4.8. Böngészésautomatizáló technológiák

A futási időben generált weblapok feldolgozásának első lépése a legoptimálisabb automatizálási megoldás kiválasztása. Ebben a feladatban több fontos szempontot is figyelembe kellett vennem a karon folyó munkába való bekapcsolódáshoz:

- Használhatónak kellett lennie Python programnyelven írt szkriptek által.
- Nyílt forráskódú licensz alatt kellett megjelennie
- Működnie kellett Linux operációs rendszeren.
- Világos dokumentációval, aktív felhasználói bázissal kellett rendelkezzen, ezzel megkönnyítve a fejlesztés folyamatát.
- Legyen *headless*, azaz legyen képes grafikus megjelenítés nélküli futásra, így gyorsítva az adatfeldolgozás folyamatát.

Ezek alapján a szempontok alapján sikerült leszűkíteni az összehasonlítandó lehetőségek körét. A következőkben röviden bemutatom és összehasonlítom a kiválogatott technológiákat (lásd. áttekintő ábra az I függelékben).

4.9. A keresőrobotok

A keresőrobot (web crawler) egy weboldalak bejárására és letöltésére létrehozott szoftver. Célja, hogy a bejárt oldalak tartalmát begyűjtse és valamilyen szempont szerint indexelje. Bár a legismertebbek keresőmotorok számára térképezik fel a weboldalakat (pl. Google, DuckDuckGo), a crawlerek felhasználási területe kiterjedt és részét képezi a nyelvtechnológiai célú felhasználás is (a dolgozatban korábban említett tartalomkinyerőket túlnyomórészt ilyen crawlerekkel párban használják). Működésüket tekintve a keresőrobotok először letöltik az aktuális weboldalt, majd a hivatkozásokat kinyerve rekurzív módon meglátogatják az összes, lapon linkelt oldalt. Több crawler feljegyzi az általa meglátogatott oldalakat, így kerülve az adatduplikációt. A keresőmotorokat kiszolgáló crawlerek működési környezetét szemlélteti a H ábra.

A linkek bejárására több forgatókönyv létezik. Kisebb mennyiségű weblap meglátogatására használt robotok használhatják az ún. kötegelt (*batch mode*) bejárást, melynek lényege, hogy a robot minden indexelési folyamatot üres tárral kezd (minden folyamat végén „elfelejti" a meglátogatott oldalakat). Így a rendelkezésre álló adatok mindig a lehető legfrissebbek, azonban nagyobb mennyiségű bejárandó weblap esetén korlátozottan alkalmazható. Ilyen esetben az inkrementális módszert szokás alkalmazni, melynek lényege, hogy a bejárt weblapokat a bejárás sorrendje (az egymásra hivatkozások) alapján a keresőrobot gráfba rendezi és az új weblapok bejárásával párhuzamosan a régieket is valamilyen metodika (mélységi-, szélességi bejárás vagy valamilyen prioritási sorrend) alapján újra és újra bejárja (frissíti) [9].

4.9.1. Selenium

A Selenium[10] egy böngészésautomatizálási keretrendszer, mely képes együttműködni a legtöbb nagy böngészővel. Bár elsősorban webalkalmazások automatizált tesztelésére fejlesztették ki és használják, mára sokkal erőteljesebb, széleskörűbb felhasználású technológiává vált. Az automatizáláshoz saját fejlesztőkörnyezettel is rendelkezik, azonban a Selenium

[10] `http://www.seleniumhq.org/`

Client API segítségével Java, C#, JavaScript, Ruby és Python nyelven írt alkalmazásokban is lehetséges keretrendszer használata.

A szoftver rendelkezik egy felhasználóbarát grafikus fejlesztőkörnyezettel (Selenium WebIDE), mely közvetlen felhasználói interakciók rögzítésének segítségével valósítja meg a szkriptek megírását: a felhasználó megnyitja a Firefox pluginként feltelepített Selenium WebIDE-t, majd egy böngészőablakban végrehajtja az automatizálni kívánt akciósorozatot. Ezt azután szövegfájlba exportálja, amely szövegfájlt a Selenium képes Python szkriptté konvertálni. Az így kapott Python nyelvű szkript ezután könnyen szerkeszthető és akár absztrahálható, hogy hasonló, de nem teljesen azonos felépítésű oldalakon is működjön.

A Selenium három rétegből épül fel:

1. A böngészőmotor és az automatizáló rendszer közötti kapcsolatot biztosító ún. webdriver,
2. maga az automatizálási rendszer, mely a legtöbb esetben fekete dobozként működik a felhasználó szemszögéből,
3. és az automatizálás testreszabására, tesztek és folyamatok megírására szolgáló API, melyen keresztül a rendszer utasításokat kap a Selenium IDE-től, vagy egy, valamelyik fentebb felsorolt programnyelven írt alkalmazástól.

A keretrendszer használható grafikus megjelenést biztosító, és headless, azaz megjelenítő nélküli böngészők (pl. Headless Chromium, PhantomJS) segítségével is. Emellett segíti a használatát, hogy nagy méretű, aktív felhasználói közösséggel rendelkezik, ami megkönnyíti a használatát ráépülő programok implementációjakor.

4.9.2. Seleniummal automatizálható böngészőmotorok

PhantomJS

A PhantomJS egy Webkit-alapú headless böngésző JavaScript API-val. Képes DOM, CSS, JSON, Canvas és SVG kezelésére, illetve képes Javascript szkriptek automatikus végrehajtására[11]. Nagy felhasználói táborral és kiváló dokumentációval rendelkezik, így könnyű

[11] `http://phantomjs.org`

utánajárni a fejlesztés során felmerülő problémáknak. Hátránya azonban, hogy fejlesztése lassú, így sok szoftverhiba javításra vár, és a böngészőmotornak is egy régebbi verzióját tartalmazza.

Headless Chromium

A Headless Chromium[12] egy Blink böngészőmotorra épülő, a Google Chrome minden technikai lehetőségét nyújtani képes headless böngésző. Webdriver segítségével Seleniumon keresztül programozható. Bár felhasználótábora kisebb, mint a PhantomJS-é, de a Chromiuméval megegyező funkcionalitás miatt egy „igazi" böngésző megbízhatóságával működik, a grafikus megjelenítés költségei nélkül. A Chrome és Chromium webdriverei azonban kevésbé rendszeresen karbantartottak, mint a többi, e fejezetben vizsgált böngészőé és Selenium és Splinter integrációja is kevésbé stabil

GeckoDriver

A GeckoDriver a Mozilla Firefox nyílt forráskódú böngésző webdrivere. Bár headless futási móddal nem rendelkezik, és így lassabb, mint a másik két, korábban bemutatott lehetőség, azonban stabil, jól karbantartott és Selenium és Splinter integráció is a kívánalmaknak megfelelően működik. Felhasználó tábora nagyobb, mint a Chrome driveré, így a felmerülő problémák is hamarabb javításra/megválaszolásra kerülnek.

4.9.3. Splinter

A Splinter[13] egy Python API-val rendelkező automatizálási rendszer, mely képes Selenium használatával – afölött absztrakciós réteget képezve – böngészésszimulálási feladatok végrehajtására. Bár lehet használni a Selenium webdriverei nélkül is, azok azonban szükségesek ahhoz, hogy az oldalak JavaScript szkriptjeit végrehajthassa. A Splinter emellett lehetővé teszi elemei Selenium API-val történő manipulációját, így a két rendszer együttes használatával kibővülnek a felhasználási lehetőségei.

[12] `https://chromium.googlesource.com/chromium/src/+/lkgr/headless/`
[13] `http://splinter.readthedocs.io/`

II. rész

Technológiai háttér

5. Meglévő technológiák elemzése – adatkinyerési módszerek

A fejlettebb, összetettebb tartalomkinyerő algoritmusok (összefoglaló a 1. táblázatban) nagy része részben vagy egészben visszavezethető bizonyos, régebben kidolgozott, kevésbé robosztus megoldásra (összefoglaló a B függelékben). Ezeket a technológiákat ma önmagukban többnyire nem használják, viszont ismeretük szükséges mind a jelenlegi tartalomkinyerők megértéséhez, mind egy esetleges új algoritmus megtervezéséhez.

5.1. Body Text Extractor algoritmusok

5.1.1. Body Text Extraction (BTE)

A BTE[14] algoritmust Aidan Finn dolgozta ki és implementálta Python nyelven 2001-ben a University College Dublin-on [10]. Az algoritmus kiválasztja a leghosszabb egybefüggő szövegrészt, míg a HTML tag-eket kiszűri. Legnagyobb hibája, hogy a dokumentumnak csak egy blokkját tudja kiemelni, így töredezett, több részből álló fő tartalmat nem tud kezelni. Később ezt az algoritmust egészítették ki a Document Slope Curve algoritmussá, majd később több más technológia alapjául szolgált (lásd B függelék).

5.1.2. Document Slope Curve

A DSP[15] 2002-ben született a BTE algoritmus továbbfejlesztéseként [11]. Bár eredetileg nem tartalomkinyerésre lett kifejlesztve, később bebizonyosodott, hogy ebben a feladatkörben is jó eredményeket ér el [12]. A módszer ablakozást használ a leghosszabb szövegblokkhoz hasonló részletek kiválasztására. Az ablakozás mellett *content code blurring* technikát is alkalmaz, hogy a túl szigorú ablakozás ne szűrjön ki lényeges tartalmat.

[14] `http://www.aidanf.net/downloads/bte_0.1.tar.gz`

[15] Az eredeti közlés inkább az elméleti háttérre helyezte a hangsúlyt, nem találtam megvalósítást.

5.1.3. Content Code Blurring

A CCB algoritmus a Body Text Extractionből Thomas Gottron által kifejlesztett módszer [13], melynek lényege, hogy merev ablakolás helyett súlyozott lokális átlagokkal számolja a HTML dokumentum egyes részeinek súlyát (C függelék). Így ha egy kisebb súlyú elem önmagában nem menne át a szűrőn, és így elveszne, ha egy nagyobb súlyú elem mellett szerepel, az átlagolás miatt mégis bekerülhet a végleges dokumentumba. Ez a módszer Gottron 2009-es mérései szerint az akkor létező algoritmusok közül a legpontosabb szűrési eredményt biztosította.

5.2. Egyéb nem összetett algoritmusok

5.2.1. Feature Extractor

A Feature Extractor egy HTML dokumentumot szűr egy kívánt tulajdonság és egy rendezett tag-halmaz alapján. Az algoritmus először partíciókra bontja a dokumentumot, majd osztályozza őket a kívánt tulajdonság előfordulása alapján. Azok a blokkok, melyben a kívánt tulajdonság előfordulási aránya nagyobb, mint a többi tulajdonság együttes előfordulása bekerülnek a *győztes halmazba*, ahonnan az a blokk lesz kiválasztva, melyben ez az arány a legnagyobb. Az algoritmus ebből a győztes blokkból nyeri ki a végső információt [14]. A Feature Extractor által használt tulajdonságok a gyakorlatban egyrészt bizonyos HTML-tagek (pl. <img>, <script>, <ul>, stb.) előfordulását jelentik, másrészt olyan jellemzőket, mint az egybefüggő szövegblokkok hossza, illetve az adott nyelveken íródott szövegekben gyakran előforduló szavak, ún. *STOPWORD*-ök aránya a szövegben (*stopWordRatio*).

5.2.2. Link Quota Filter

Az LQF egy elterjedt heurisztikus módszer navigációs menük és link listák szűrésére. A technológia lényege, hogy megtalálja azokat az elemeket, melyekben nagy arányban találhatók hivatkozások. Az LQF egy fejlett változatát Constantine Mantratzis mutatta be 2005-ben [15]. Ez a módszer csak a hivatkozás típusú boilerplate tartalmakat tudja kiszűrni [16].

1. táblázat. Tartalomkinyerő algoritmusok

Algoritmus	**Forrás**	**Kiválasztott blokk**
Body Text Extraction	DOM-fa	Leghosszabb egybefüggő szövegrésszel rendelkező blokk.
Document Slope Curve	HTML kód	A dokumentum azon részei, melyek alacsony HTML tag-aránnyal rendelkeznek
Content Code Blurring	HTML kód	Azok a blokkok, melyek elég nagy tartalom-kód aránnyal rendelkeznek. Lokális átlagokat számol a blokkok kiválasztásához
Feature Extractor	DOM fa	Azon blokkok melyek a megadott tulajdonsággal leginkább rendelkeznek.
Link Quota Filter	DOM fa	Kiszűri a magas hivatkozás-szöveg aránnyal rendelkező blokkokat.
Largest Pagelet	DOM fa	Legmagasabb látható szöveg tartalommal rendelkező partíció.

5.2.3. Largest Pagelet

A Largest Pagelet algoritmust Sridhar Rajagopalan és Ziv Bar-Yossef dolgozta ki 2005-ben [12]. A módszer lényege, hogy a HTML dokumentumot elkülönülő részekre *(pagelet)* osztja, majd kiválasztja a legtöbb szöveget tartalmazó pageletet. Ez a legnagyobb blokk, a *largest pagelet* lesz a fő tartalomként kiválasztott része a dokumentumnak.

5.3. Readability

A Readability[16] egy nem tudományos kutatásokon alapuló tartalomkinyerő rendszer, melyet az Arc90 dizájner csapat hozott létre kísérletképpen 2009-ben. Jelenleg[17] a Mozilla Firefox böngésző olvasási nézete mögött a Readability fut. A rendszer mögött futó algoritmus sokszor verziószámonként változott, nem kutatások és elemzések, hanem inkább intuitív kísérletezés eredményeként. A jelenlegi verzió a megtekintett weblapba illesztett JavaScript kódból áll, mely a DOM-fával dolgozik, eltávolítja a boilerplate elemeket a dokumentumból.

A szkript akkor kerül a HTML kódba, amikor a felhasználó bekapcsolja az olvasási

[16] Az eredeti, arc90-féle verzió: `https://github.com/MHordecki/readability-redux` A legfrissebb Mozilla-féle verzió: `https://github.com/mozilla/readability`

[17] A 38.0.5-ös verziótól kezdve. `https://www.mozilla.org/en-US/firefox/38.0.5/releasenotes/`

nézetet a böngészőben. Az algoritmus több összeillesztett heurisztikából és szabályból áll. A Readability-t népszerűsége miatt több más nyelvre, így Pythonra[18], NodeJs-re[19] Java-ra, C#-ra és Ruby-ra[20] is portolták. Ezek a portok Tomaž Kovačič mérései szerint változó minőségűek [17] és parancssori alkalmazásként futtathatók.

Az algoritmus (1) első lépése, hogy beolvassa egy DOM-fába a HTML-dokumentumot. Második lépésben minden *script* node-ot eltávolít, majd kibányássza a címet a <title> tagből. Harmadik lépésben egy előre definiált lista alapján szűri a node-okat. Ez a lista tartalmazza azokat a HTML-tagekből olvasott DOM node-okat, melyek az algoritmus készítői szerint nem tartalmaznak információt. Ezután egy ciklussal végigmenve az összes bekezdésen (<p> tag) egy pontszámot rendel hozzájuk a szöveges tartalom arányában. Ezeket a pontszámokat átadjuk az összes szülőelemnek. Az algoritmus ezután kiválasztja a legnagyobb pontszámú konténert (top candidate). Ha a top candidate szöveges tartalma kevesebb, mint

[18] `https://github.com/gfxmonk/python-readability`
[19] https://github.com/arrix/node-readability
[20] `https://github.com/cantino/ruby-readability`

250 karakter, akkor igazított paraméterekkel újra lefut az algoritmus [17].

Data: h := HTML forráskód;
dom:= a HTML kódból kinyert DOM-fa;
deleteScripts(dom) `/* <script> node-ok eltávolítása a fából */`
findTitle(dom) `/* cím kibányászása a <title> tag-ből heurisztikus módszerrel */`
clearNodes(dom) `/* egy előre definiált statikus node lista alapján szűri a fa node-jait */`
foreach *paragraph p in dom* **do**
    p.score := scoreP(p) pontok hozzárendelése a bekezdésekhez;
    parent := getParentNode(p);
    parent.score := parent.score+p.score;
    grandParent := getGrandParentNode(p);
    grandParent.score := grandParent.score+(p.score/2) ;
end
chooseTopCandidate() `/* legmagasabb pontszámú jelölt kiválasztása */`
topCandidate = findSiblingContent(topCandidate) `/* a top candidate tartalmának kiolvasása */`
if *length(topCanidate.text) <250* **then**
    algoritmus újrafuttatása igazított paraméterekkel
else
    return *<title, topCandidate.text >*
end

1. algoritmus: Readability algoritmus

5.4. Több algoritmust kombináló adatkinyerő megoldások

5.4.1. Crunch

A Crunch[21] algoritmust Suhit Gupta fejlesztette ki, és mutatta be a Columbia University-n 2002 szeptemberében [18]. Az algoritmus a HTML kód helyett az abból (1.0-ás verzióban openXML-el[22], később más XML parserekkel felépített) felépített DOM-treet használja a releváns tartalom kinyerésére.

A Crunch algoritmus lépései röviden: végigolvassa a HTML-t, felépíti a DOM-tree-t, rekurzívan végigjárja a fa node-jait, és kiszűri a zajosnak ítélt tartalmakat. A Crunch jellemzően két féle szűrőt használ a tartalom szűrésére: az első egyszerűen eltávolít bizonyos HTML elemeket, pl. az *image*, a *style* és *link* tageket. A második szűrő eltávolítja a hirdetéseket egy statikus szerver-címlista lista alapján, az üres *table*-ket és a *link list*-eket. Utóbbiakat az adott DOM-node *link-to-text-ratio*-jának kiszámolásával (a Link Quota Filter algoritmussal, ld. 5.2.2) elemzi és bírálja el. Ha a link-to-text ratio túl magas, akkor a tartalmat zajosnak ítéli meg, és kiszűri. A szűrés küszöbe az algoritmusban átállítható. Ezen kívül a Crunch magába foglal egy *link retainer*-t, ami a szűrt weblap végén kilistázza a kiszűrt linkeket, hogy a weblap böngészhetőségét megtartsa. Gupta a Crunch három verzióját készítette el, én most a legújabb, a 3.0-s verziót elemzem.

A Crunch egy proxy szerverre (lásd D függelék) irányítja át a bejövő HTTP tartalmakat, mely az algoritmus segítségével megszűri, majd a böngészőnek továbbküldi az oldalak HTML-kódját. A proxy egy grafikus felhasználói felületen keresztül (is) konfigurálható, újabb algoritmusokkal és modulokkal bővíthető.

Az algoritmus egyik legnagyobb előnye a többi tartalomkinyerő algoritmushoz képest az, hogy képes weblapokat osztályozni, típusokba rendezni, majd a típusuk alapján különböző eljárásokkal szűrni. Ilyen osztályok például a hírlapok, webshopok, stb. Gupta terveiben szerepelt, hogy a Crunch-ot kiegészítsék egy gépi tanuló algoritmussal a típusfelismerés mellett, ám a dolgozat megírásáig ez nem történt meg.

Mivel a Crunch nem csak egy olvasási nézetet hoz létre, hanem egy teljes böngészé-

[21] `http://code.psl.cs.columbia.edu/?a=summary&p=crunch3`

[22] Nem a Microsoft dokumentum-szabvány, hanem egy, az eredeti közlésben nem hivatkozott XML parser.

si folyamatot manipulál, így lényeges, hogy ne szűrje ki a releváns linkeket, és azokat az információkat, amelyekre a felhasználó böngészés közben kíváncsi lehet. A teljes böngészési folyamatok manipulálása miatt fontos szempont a használhatóság szempontjából, hogy megtartsa a fontos linkeket, illetve hogy lehetséges legyen vele tájékozódni és mozogni a weblapokon. Ezeket a szempontokat a Crunch figyelembe vette és teljesítette. A fejlesztők tesztjei alapján a képernyőolvasóval böngésző tesztalany könnyebben és gyorsabban teljesítette a kapott feladatokat Crunch használatakor, mint anélkül [3].

5.4.2. CombinE System

A CombinE rendszert Thomas Gottron dolgozta ki 2008-ban a Mainz-i egyetemen [19]. A rendszer alapelgondolása az, hogy a tartalomkinyerő algoritmusok kombinálásával és egyidejű használatával pontosabb szűrést lehet végezni, mint ha csak kiválasztjuk a legmegfelelőbb algoritmust. A CombinE egy proxy szerveren működve szűri meg a letöltött adatokat, majd továbbítja azokat a felhasználó számítógépére. A dokumentumok szűréséhez a rendszer szűrőalgoritmusok bővíthető halmazát használja, melyek különböző módokon lehetnek egymással kombinálva (lásd E függelék). A kombinálás történhet soros és párhuzamos működéssel, valamint ún. szavazásos (voting) rendszer szerint. A soros rendszer implementálása egyszerű, az egyik algoritmus kimenete a másik algoritmus bemenete. A második, párhuzamos működésű rendszernél az algoritmusok az eredeti dokumentum egy-egy példányán dolgoznak, majd a kimenetekből vagy metszetet, vagy uniót alkotva állítják össze a végleges eredményt. A szavazásos rendszer gyakorlatilag a párhuzamos megvalósítás egy speciális változata: az algoritmusok az eredeti példányon dolgozva, annak egyes elemeiről szavaznak, hogy benne legyen-e a végleges kimenetben? Ha a dokumentum ezen része elér egy bizonyos szavazatszámot, akkor a main content részeként lesz kezelve, ha nem, akkor ki lesz szűrve.

A CombinE nem a HTML kódot, hanem az abból előállított DOM-fát használja a HTML dokumentumok feldolgozására. A tartalom szűrésének lépései:

1. a HTML dokumentum normalizálása
2. a HTML kód szűrése a beépített algoritmusokkal (a kimenet egy N elemű, szűrt HTML

kódokból álló tömb)

3. a tömb elemeinek feldolgozása egy DOM-fába, majd a fák egyesítése (a kiválasztott módszerrel)
4. HTML kód készítése a DOM-fából
5. a HTML dokumentum elküldése a felhasználónak

5.4.3. JusText

A JusText-et[23] (lásd F függelék) Jan Pomikálek mutatta be 2011-ben doktori dolgozatában a Masaryk University-n [1]. A JusText elsősorban nyelvészeti célú felhasználásra tervezett szoftver. Az algoritmus négy fő szakaszra különül el:

- **Szegmentálás** A JusText első lépésben feldarabolja az oldalt adatkinyerés szempontjából nagyjából homogén blokkokra. A boilerplate és az értékes tartalmak is többnyire csoportokba vannak rendeződve. Egy boilerplate-et tartalmazó blokk legtöbbször boilerplate-tel van körülvéve, az értékes blokkok pedig további értékes blokkokkal. Az alapgondolat az, hogy a legtöbb hosszú és néhány rövid blokkról könnyen eldönthető, hogy boilerplate-e, az eldöntetleneknél pedig azt nézzük, hogy milyen címkéjű blokkok veszik őket körül. Ezt a folyamatot az algoritmus következő irányelvek segítségével hajtja végre:
  - A rövid, linket tartalmazó, illetve a bármilyen hosszú, de sok linket magukba foglaló blokkok szinte mindig boilerplate tartalmat jelentenek.
  - A hosszú, összefüggő szöveget tartalmazó blokkok általában jó, de a hosszú nem összefüggő szöveget tartalmazók általában boilerplate tartalmat takarnak.

  Az összefüggő szöveg elemzése öt paraméter alapján történik. Ezek a paraméterek nem egyszerű ablakozással, hanem súlyozva kerülnek számításba az osztályozáskor (lásd 2. táblázat). A blokkok szűrésekor használt paraméterek:

[23] `http://corpus.tools/wiki/Justext`

– MAX_LINK_DENSITY – a szövegben előforduló linkek aránya a teljes szöveghez képest. Alapértéke: 0.2

– LENGTH_LOW – a releváns szöveg minimum hossza. Alapértéke: 10

– STOPWORD_LOW – a szövegben előforduló *stopword*-ök minimum aránya. Alapértéke: 0.3

– LENGTH_HIGH – a releváns szöveg maximum hossza. Alapértéke: 30

– STOPWORDS_HIGH – a szövegben előforduló *stopword*-ök minimum aránya. Alapértéke: 0.32

2. táblázat. A blokkok besorolása a *stopword* arány és a szöveg hossza alapján

blokk hossza	**stopword sűrűség**	**végleges osztály**
közepesen hosszú	alacsony	rossz
hosszú	alacsony	rossz
közepesen hosszú	közepes	majdnem jó
hosszú	közepes	majdnem jó
közepesen hosszú	nagy	majdnem jó
hosszú	nagy	jó

- **Előfeldolgozás** Ebben a szakaszban eltávolításra kerülnek a <header>, <style> és <script> tagek. A Victor algoritmussal (lásd 5.4.5. fejezet) való kompatibilitás jegyében beállítható, hogy a <select> tagek is törölve legyenek. Ha ez nincs beállítva, a <select> tagek akkor is automatikusan a boilerplate kategóriába kerülnek. Emellett a © szimbólumot tartalmazó blokkok is boilerplate címkét kapnak[24]. Az algoritmus pszeudokódja:

[24] Ez nem ugyanaz, mint az eltávolítás, a JusText ugyanis nem törli ki, hanem csak megcímkézi a boilerplate tartalmakat.

```
if linkDensity >MAX_LINK_DENSITY then
    return bad
end
if tokenCount <LENGTH_LOW then
    if linkDensity > 0 then
        return bad
    else
        return short
    end
end
if stopWordDensity >STOPWORD_HIGH then
    if tokenCount >LENGTH_HIGH then
        return good
    else
        return near good
    end
end
if stopWordDensity >STOPWORD_LOW then
    return near good
else
    return bad
end
```

2. algoritmus: A HTML-dokumentum előfeldolgozása

- **Környezetfüggetlen osztályozás** Ebben a lépésben az algoritmus négy felé osztályozza a blokkokat:
  - **Jó** – értékes blokkok
  - **Rossz** – boilerplate blokkok
  - **Rövid** – nem eldönthető (túl rövid)
  - **Majdnem jó** – a rövid és a jó között
- **Környezetfüggő osztályozás** Itt az osztályozott blokkokat *majdnem jó* és *rövid* blok-

kok sorozataként nézzük, melyeket *jó* és *rossz* blokkok határolnak. Ekkor ha egy blokkot két *jó* blokk határol, akkor *jó* besorolást kap, ha két *rossz*, akkor *rosszat*. A problémás esetek azok, amikor egy *jó* és egy *rossz* blokk határolja a *majdnem jó* és *rövid*. Ekkor minden, egy *majdnem jó* és egy *rossz* blokk közé eső blokkot *rossz*, az összes többit *jó* címkével látjuk el (lásd 3. algoritmus).

```
Data: block := elemzett blokk;
if block == bad then
    return bad;
end
if block == good then
    return good ;
end
if block.pre_class == "near-good" then
    prevClass = get_class_of_prev_good_or_bad_block(block) nextClass =
     get_class_of_next_good_or_bad_block(block) if prevClass == "good" or
     nextClass == "good" then
        return good
    else
        return bad
    end
end
if block.pre_class == 'short' then
    prevClass = getClassOfPrevGoodOrBadBlock(block);
    nextClass = getClassOfNextGoodOrBadBlock(block) ;
    if prevClass == 'bad' and nextClass == 'bad' then
        return bad
    end
    if prevClass == 'good' and nextClass == 'good' then
        return good
    end
    if prevClass == 'bad' and nextClass == 'good' then
        if getClassOfPrevNonShortBlock(block) == 'near-good' then
            return good
        else
            return bad
        end
    end
    if nextClass == 'bad' and prevClass == 'good' then
        if getClassOfNextNonShortBlock(block) == 'near-good' then
            return good
        else
            return bad
        end
    end
end
```

3. algoritmus: Környezetfüggő osztályozás

A blokkok besorolása az osztályokba a *stopWordRatio* (lásd 5.2.1. fejezet) és a Document Slope Curve (lásd 5.1.2. fejezet) algoritmus elve alapján történik (lásd 2.

algoritmus).

5.4.4. boilerpipe

A boilerpipe[25] egy döntési fákon alapuló tartalomkinyerő algoritmus-család, amit elsősorban cikkek híroldalakból történő kinyerésére fejlesztett ki Christian Kohlschütter [20]. A technológia egy Java könyvtárként van megvalósítva és alapvetően két tulajdonságot használ fel a lényeges tartalom kinyerésére: az egybefüggő szöveg hosszát és sűrűségét. A boilerpipe DefaultExtractor tartalomosztályzó algotitmusa:

```
if currentLinkDensity<=0.333333:
    if previousLinkDensity<=0.555556:
        if currentNumWords<=16:
            if nextNumWords<=15:
                if previousNumWords<=4:
                    return boilerplate
                else:
                    return content
            else:
                return content
        else:
            return content
    else:
        if currentNumWords<=40:
            if nextNumWords<=17:
                return boilerplate
            else:
                return content
        else:
            return content
else:
    return boilerplate
```

4. algoritmus: A Boilerpipe DefaultExtractor algoritmusa

5.4.5. Victor

A Victor[26] egy felügyelt gépi tanuló algoritmussal dolgozó, Perlben implementált tartalomkinyerő algoritmus [21]. A dokumentum letöltése után először ellenőrzi annak nyelvét és az

[25] `http://code.google.com/p/boilerpipe/`
[26] `http://ufal.mff.cuni.cz/victor/`

esetleges érvénytelen karakterek jelenlétét. Ha azt érzékeli, hogy valamelyik a kettő közül nem megfelelő (ez az eredeti implementációban a cseh nyelvet és/vagy rossz karakterkódolást jelent), akkor a dokumentumot eldobja. Ezután a HTML kód hibáit a Tidy[27] algoritmussal kijavítja és az ilyen módon kapott kódból kiszűri az értékes szöveget nem tartalmazó elemeket (<script> és <style> tagek, beszúrt külső objektumok). A következő lépésben a megtisztított HTML-ben a szöveges tartalmat blokkok sorozatára osztja ahol a HTML tagek jelentik a blokkok határát. Pl. a <h1>Hello world again!</h1> HTML kód esetében a szöveget a 'Hello' és a 'world again'! blokkokra osztaná. Ezt követően minden egyes blokkhoz hozzárendel bizonyos tulajdonságokat és manuálisan a következő címkék valamelyikével látja el őket:

- header - blokkcímek
- text - ez a releváns tartalom
- other - ez a boilerplate

Az algoritmus ilyenkor megjegyzi az egyes címkékhez rendelt tulajdonságokat és azok értékét, így egyre inkább hatékonnyá válik a blokkok automatizált címkézésére. Az algoritmus végül a *header* és a *text* blokkokat bennhagyja a szűrt dokumentumban, míg az *other* címkével ellátottakat kiszűri. A tanuló algoritmust 51 HTML dokumentumon futtatták le, manuálisan igazítva a CleanEval irányelvekhez.

5.5. A dokumentum-szintű metódusok előnyei

A dokumentum-szintű metódusok több előnnyel is rendelkeznek a webhely-szintű megoldásokkal szemben. Az egyik ilyen, hogy függetlenek a webhely felépítésétől, nem zavarja őket, ha nincs egy, a weboldal összes dokumentumán átnyúló sablon. Emellett nagyobb múltra tekintenek vissza, így sok esetben igen kifinomult megoldások születtek a korábban már kidolgozott algoritmusokra alapozva. További érv a dokumentum-szinten működő technológiák használata mellett, hogy nincs hidegindítási idejük, így könnyen alkalmazhatók

[27] `http://tidy.sourceforge.net/`

felhasználók számára készült alkalmazásokban, például olvasási vagy akadálymentesített nézet létrehozására. Könnyen futtathatók egy proxy-szerveren, ami viszonylag gyorsan szűrve a kívánt oldalt rugalmas felhasználást biztosít a felhasználóknak.

5.6. A dokumentum-szintű metódusok hibái

A dokumentum-szintű technológiák ugyanakkor több szempontból kevésbé robusztusak, mint a webhely-szintűek. Nem tudják ugyanis kihasználni a modern webes technológiák egy részét, mint a tartalomkezelő rendszerek által létrehozott oldalsablonok (*template-k*) jelenlétét a webhelyeken. Emellett minden weblap-felépítésre külön kell optimalizálni őket, hogy ideális eredményt hozzanak különböző struktúrájú dokumentumokon (ezt a problémát igyekeznek megoldani a több dokumentum-szintű algoritmust kombináló összetett technológiák, pl. Crunch, CombinE System).

6. Webhely-szintű algoritmusok

6.1. Bevezetés

A dinamikus webhelyek[28] elterjedésével egyre inkább megfigyelhető lett, hogy a weblapok adott sablonok alapján épülnek fel, melyek egy weboldal esetén azonos helyen tartalmazzák az azonos funkciójú blokkokat (pl. navigációs menü, hirdetések, fő tartalom). Az ilyen sablonok felismerésére különböző algoritmusok jöttek létre, melyek nagy hatásfokkal tudják kiszűrni a weblapok boilerplate tartalmát. Az ilyen technológiák azonban legtöbbször crawlerekkel működnek, melyek működését több esetben akadályozhatják különböző feldolgozni kívánt weboldalak. A módszerben rejlő lehetőség, hogy felismert sablonok a tartalomkezelő rendszerek korlátai miatt sokszor különböző weblapokon is azonosak vagy hasonlóak, így tanuló algoritmusokkal viszonylag pontos szűrési eredményeket lehet elérni akár egy weblap feldolgozása után is. A módszernek ugyanakkor kihívása a honlapok belső hivatkozásainak felismerése, ami fontos a sablonok pontos betanulásához.

[28] A félreértések elkerülése végett az alábbi fejezetben a böngészők által letöltött HTML fájlokat weblapnak vagy weboldalnak, ezek, felhasználók számára hierarchiába rendezett csoportját webhelynek vagy honlapnak nevezem.

6.2. Bar-Yossef és Rajagopalan

Ziv Bar-Yossef és Sradhar Rajagopalan 2002-ben dolgozták ki weblap-szintű sablondetektáló algoritmusukat [22]. Módszerük lényege, hogy a bejárt dokumentumokat a linkek száma alapján Broder algoritmusával[29] megkeresik a megegyező vagy hasonló HTML-blokkokat, majd azokat a blokkokat, melyek lenyomatával több más blokk lenyomata is megegyezik csoportokba osztják. Ezután az egynél több tagú csoportokhoz kilistázzák az összes weblapot, amelyik legalább egy blokkot tartalmaz a csoportból. Az így kialakult weblap-csoportokat összekötő hivatkozásokat kielemzik, és a csoportot egy irányítatlan gráfként ábrázolják. Végül ezt a gráfot darabokra bontják, és az összes, egynél több weboldalban jelen levő blokkot boilerplate-nek jelölik.

6.3. Site Style Tree

A *Site Style Tree* metódust Lan Yi, Bing Liu és Xiaoli Li fejlesztette ki az internetes adatbányászat megkönnyítése céljából [24]. Az technológia alap gondolata az, hogy felépítenek egy DOM fát a megvizsgált weblapokból, majd ezeket összefésülik (lásd G függelék). Az összefésülés úgy történik, hogy a különböző oldalakban szereplő azonos HTML-blokkokat egy csúcsba helyezik, megjegyzik a csúcsba összegyűjtött blokkok számát, majd folytatják az összefésülést a csúcs gyermekeivel. A fa magas blokkszámú csúcsai gyakran ismétlődő tartalmakat takarnak, ami a boilerplate tartalmak jellemző tulajdonsága.

6.4. David Gibson algoritmusa

David Gibson, az IBM kutatója 2005-ben mutatta be sablondetektáló algoritmusát [25]. Ez a technológia szintén a DOM-fán alapul. Az oldalak DOM-fáinak összes részfájából lenyomatot képez, majd a megszámolja az azonos lenyomatokat. Ha egy lenyomat egy bizonyos alsó és felső határ között fordul elő az oldalakon, akkor az algoritmus boilerplatenek jelöli és kiszűri. A felső határ azért kell, hogy bizonyos HTML-tagek, melyek gyakran

[29] Andrei Z. Broder 2000-ben mutatta be duplikáció-szűrő algoritmusát [23]. A módszer lényege, hogy a HTML kódot ún. *pageletekre* osztja, melyekből bizonyos tulajdonságok (pl. szöveges tartalom, linkek aránya) alapján lenyomatot képez, majd összegyűjti az azonos vagy nagyon hasonló lenyomatokat.

előfordulhatnak az oldalakon, de nem jelentenek boilerplate-et (pl.
) ne kerüljenek szűrésre. Ugyanez a csapat később egy másik megoldást is kidolgozott, melyben elhagyták a HTML tageket, és csak a szövegek előfordulása alapján szűrték a tartalmat. Ez a módszer sokkal gyorsabbnak, és ugyanolyan pontosnak bizonyult.

6.5. Goldminer

A Goldminer algoritmust Endrédy István és Novák Attila fejlesztette ki 2013-ban [26]. A metódus célja a JusText algoritmus hatékonyságának növelése összefüggő szövegek gyűjtése esetén webhely-szintű megoldások alkalmazásával. A JusText ugyanis, bár viszonylag nagy pontossággal szűri ki a releváns tartalmat a weboldalakról, sokszor tönkreteszi a szövegek koherenciáját, mivel kiszűri a duplikációnak vélt, sokszor szereplő mondatokat és bekezdéseket.

Az algoritmus először letölt egy oldalt a webhelyről, majd ezen felismeri az oldalspecifikus HTML-sablont. Ezután bejár mintegy száz oldalt a webhelyről, melyeken lefuttatja a JusText algoritmust. Ez „jó" és „rossz" osztályba sorolja az oldal egyes részeit. Ezután összegyűjti a „jó" blokkokat az összes általa bejárt oldalról, és azokat, amelyek több oldalon is előfordulnak átminősíti „rossz" blokkokká. A következő lépésben az algoritmus megkeresi a „jó" blokkok legközelebbi közös ősét minden egyes oldalon. A tanulófázis végén a leggyakoribb közös szülő lesz győztesnek jelölve. Ezután még egy keresés lefut a győztes blokkon belül, ami módosíthatja a győztes blokk határait, így lezárva a tanulófázist. Ezután az algoritmus az adott webhelyet bejárva mindig csak a webhelyspecifikus kezdő- és végpontok közötti oldalrészletet adja át a JusTextnek szűrésre.

6.6. A webhely-szintű metódusok előnyei

A webhely-szintű technológiák ereje abban rejlik, hogy ki tudják használni azt, hogy a modern tartalomkezelő rendszereket használó webhelyek legtöbb oldala azonos sablon alapján épül fel. Ekkor ezek az algoritmusok ezt a sablont kiszűrve viszonylag tiszta eredményt tudnak produkálni a vizsgált webhelyeken. Ilyenkor, mivel minden új oldalnál újra megvizsgálják annak felépítését, emiatt nem kell minden sablontípusra külön optimalizálni őket

- egy algoritmus mindenféle oldalfelépítést felismer, általánosan használható. Nyelvészeti célú alkalmazás esetén a webhelyszintű metódusok hatékonyságát növelik az automatizált keresőrobot (crawlerek), míg ez a dokumentum-szinten szűrő algoritmusoknál indifferens.

6.7. A webhely-szintű metódusok hibái

A honlap-szintű algoritmusoknak több komoly gyengesége is van, ezek alkalmazási terület szerint két irányból is megközelíthetők: a nyelvészeti célú, illetve a felhasználói célú tartalomszűrés szempontjából. Bármelyik irányból nézzük, a sablondetektáló algoritmusok nem képesek felismerni a sablonon kívüli, csak egy oldalon megjelenő boilerplate elemeket, így meghagyhatnak olyan részeket, melyek nem tartalmaznak egész mondatokat: link listák, releváns linkek, hirdetések stb. Emellett egy oldalon belül is megváltozhat annyira a sablon, hogy az az algoritmust megzavarja, így rontva a szűrés pontosságát.

Probléma az is, hogy hogyan szerzik be a megfelelő számú weboldalt a webhelyről. Egy keresőrobot be tud járni több oldalt is, ám sok esetben nagy tárhelyre lenne szükség ahhoz, hogy egy ilyen technológia hatékonyan működhessen, illetve, ha a felhasználói célú alkalmazást tekintjük, akkor nem is biztos, hogy már valaha elemezte a látogatott webhelyet. Ebben az esetben vagy csak lassan tudja előállítani a kívánt eredményt és a felhasználónak meg kell várnia az összes szükséges oldal letöltését és elemzését, vagy a metódus nem szűri ki a nem kívánt tartalmakat. A hidegindítási idő függ az internetkapcsolat sebességétől és az alkalmazott szűrőalgoritmus futási idejétől.

Látható tehát, hogy önmagukban alkalmazva a webhely-szintű metódusok csak bizonyos esetekben képesek a kellő pontosságot elérni, és működésük a legtöbb esetben komplexebb, mint a weboldal-szintű megoldásoké.

7. Következtetések a környezet kutatásából

A webhely- és weboldal-szintű technológiák kombinálására jó példa a Goldminer algoritmus, ami a JusText metódust alkalmazza az általa kiszűrt sablonok alapján felépülő weblapokon. Ez a megoldás nagyon pontos eredményt képes elérni a legtöbb esetben, de a futási idejében

összeadódik a webhely-szintű és a weblap-szintű algoritmus futási ideje.

A bevezetésben először röviden bemutattam a modern webes környezetet és technológiákat, betekintettem annak kihívásaiba és lehetőségeibe tartalomkinyerési szempontból. Ezután felsoroltam a tartalomkinyerő algoritmusok néhány lehetséges felhasználási módját, áttekintettem néhányat az ismertebb tartalomkinyerő algoritmusok közül és összehasonlítottam őket működési elvük alapján. Megállapítható, hogy bár a meglévő technológiák sok esetben nagy pontossággal ki tudják szűrni azokat az összefüggő szövegeket melyek nyelvészeti felhasználás céljából hasznosak lehetnek, azonban legtöbbjük csak bizonyos kialakítással rendelkező webhelyeken működik, vagy csak nyelvészeti célú adatgyűjtésre használható.

Minőségi ugrást jelentett a tartalomkinyerő algoritmusokban a gépi tanuló algoritmusok alkalmazása. Ezen technológiák implementálása miatt a vizsgált oldalak számának növekedésével a tartalomkinyerés egyre pontosabbá válik, így a dokumentum-szintű technológiáknál alkalmazott kézi paraméterérték-beállítás helyett a gépi tanuló algoritmus képes a lehető legoptimálisabb értékre beállítani az algoritmusban alkalmazott paramétereket. Ez nagy mennyiségű oldal megvizsgálása esetén különösen hatékony lehet (több technológia kezdetben felügyelt tanuló algoritmust használ, ami bár növeli a manuális munka mennyiségét, növelni tudja annak tanulási pontosságát).

A dolgozat további részében a bemutatott tartalomkinyerő technológiák alkalmazhatóságát vizsgálom erősen szkriptelt, futásidőben generált weblapokon, illetve javaslatot fogalmazok meg ezen területen történő felhasználásuk fejlesztésére, optimalizálására. Ezen optimalizálás alapgondolata a felhasználók viselkedésének szimulálása böngészésautomatizáló technológiák alkalmazásával. A következő fejezetben e technológiákat veszem sorra, mutatom be röviden és bemutatom azt a state-of-the-art megoldást is, melyet végül a feladat megoldására felhasználtam.

III. rész

A munka ismertetése

8. A jelenkori technológiák kihívásai futásidőben generált weblapok feldolgozásakor

A statikus weblapokra tervezett adatkinyerési technológiák - az oldalakon végigfutó keresőrobotok és a letöltött oldalakat megtisztító tartalomkinyerők - alkalmazása dinamikus környezetben több szempontból is problematikus. Ezek a problémák egyrészt közösek, másrészt eltérőek: közös probléma például, hogy a szkriptvégrehajtás nélkül elérhetetlen oldalrészek rejthetnek értékes szövegeket vagy a keresőrobotok számára bejárható linkeket, speciálisan a tartalomkinyerőké pedig a rövid szövegek és a boilerplate nehéz megkülönböztethetősége.

8.1. A tartalomkinyerő rendszerek kihívásai

A hagyományos tartalomkinyerő technológiák alkalmazásának ebben a környezetben két fő akadálya van. Az első, hogy, míg azok a weblapok, melyekre ezek a módszerek specializálódtak hosszabb, egybefüggő szövegek bányászatára lettek kifejlesztve, addig az új típusú, gyorsan frissülő oldalak és a közösségi média sokszor rövid felhasználói üzeneteket, hozzászólásokat közvetít. Emiatt a Body Text Extraction, Content Code Blurring és Largest Pagelet alapú algoritmusok nem alkalmasak a releváns tartalom elkülönítésére a gépek által előállított boilerplate-től.

A másik, hogy ezeknek az oldalaknak a kliensoldali része JavaScript nyelven van implementálva, így bányászatukhoz szükséges ezen szkriptek végrehajtása. Ez a szkriptvégrehajtást nem biztosító, vagy kifejezetten blokkoló hagyományos tartalomkinyerő technológiákkal nem lehetséges, így e dolgozat egyik első célja az volt, hogy felmérje, milyen lehetőségek állnak rendelkezésre JavaScript szkriptek gyors végrehajtására, és az általuk indított adatlekérések végrehajtására.

Az oldalak frissítése és új adatok letöltése két módon történhet meg: vagy automati-

kusan, felhasználói közbeavatkozás nélkül (pl. Facebook), vagy egy értesítés felbukkanása után, a felhasználó által kiadott utasítás hatására (pl. Twitter, Index.hu). Egy tartalomkinyerő technológiának mindkét esetben működnie kell, így fel kell ismernie, hogy az adott oldalon milyen akciókat kell végrehajtania az új tartalmak lehívásához. Ez a tanulási folyamat működhet felügyelet tanulással vagy önállóan ezt az implementáció során szükséges eldönteni.

8.2. A keresőrobotok kihívásai

A 4.9 fejezetben tárgyalt web crawlerek (keresőrobotok) problémája kisebb, mint a tartalomkinyerőké. A legfőbb kihívás a felhasználói interakció, illetve kliensoldali szkriptvégrehajtás nélkül elérhetetlen oldalrészek felderítése a tartalomkinyerők számára, valamint az ezeken az oldalrészeken feltárt linkek bejárása. Ez a felhasználói interakció lehet egy szkript végrehajtását elindító elem megnyomása (legtöbbször JavaScript void linkek), másrészt görgetés, amire reagálva az oldal alján új tartalmak jelennek meg.

Ezen problémák elhárításában lehet kiemelkedő szerepe a dolgozatban korábban tárgyalt böngészőautomatizálási rendszereknek, mint pl. a Seleniumnak, melyekkel a felhasználó által megadott interakciók absztrahálásával a crawler megtanulhatja bejárni az oldal általa egészen addig felderítetlen részét. Ha ezt a problémát sikerül megoldani, akkor a tartalomkinyerő rendszerek egyik problémája is megoldódik, így egyedül a rövid szövegek kinyerését hagyva hátra feladatnak.

A dolgozat további részében ezen probléma megoldására teszek kísérletet egy Python nyelven implementált, dinamikus oldalakat felhasználó által definiált akciók sorozatával bejáró crawler implementálásával. Ezután értékelem a crawler teljesítményét és javaslatot tesz annak további fejlesztésére.

9. A dinamikus környezetben működő keresőrobot

9.1. Mesterséges intelligencia használata oldalelemek osztályozására

A dinamikusan generált környezet kihívásainak megoldására az új keresőrobot egy döntési fát használ az egyes oldalelemek és hivatkozások osztályozására. A fejezetben a hivatkozás kifejezés alatt nem kizárólag a HTML <a> elemet értem, hanem bármely olyan oldalelemet, ami:

1. más HTML dokumentumra átirányít
2. ugyanennek az oldalnak más szekciójára átirányít
3. az aktuális oldalnak egy részét módosítja vagy elérhetővé teszi

Ez a döntési fa egy tanulóhalmazon történő tanulási folyamat után osztályokra bontja a weboldalon talált hivatkozásokat. A mesterséges intelligenciával kiegészített crawlerek így végigjárhatják a dinamikus oldalakat, automatikusan osztályozva az oldalelemeket, megjelenítve az addig rejtett tartalmakat a kinyerőalgoritmusok számára és az eddigieknél kiterjedtebb és összetettebb hivatkozáslistákat készíthetnek, így biztosítva a crawling finomhangolásának lehetőségét. A crawler számára érdekes elemeket tartalmazó előre definiált osztályok a következők:

SectionRevealer: ebbe az osztályba kerülnek azok a hivatkozások, melyek megnyomásukkor egy addig rejtett részét jelenítik meg a honlapnak. Ezek általában JavaScript void függvényekre mutató hivatkozások *void(0)* vagy # href attribútummal. A Facebook „Korábbi kommentek...” linkje például így néz ki:

```
<a class="UFIPagerLink" href="#" role="button">
  <em class="_4qba"
   data-intl-translation="View previous replies">
    <!-- react-text: 88 -->
     View previous replies
```

```
    <!-- /react-text -->
  </em>
</a>
```

A Twitter viszont hivatkozás (<a>) helyett elemeket használ:

```
<div class="tweet js-stream-tweet
            js-actionable-tweet
            js-profile-popup-actionable
            original-tweet
            js-original-tweet
            has-cards  has-content"
   data-tweet-id="838654147184496640"
   data-item-id="838654147184496640"
   data-permalink-path="/Reuters/status/838654147184496640"
   data-conversation-id="838654147184496640"
   data-tweet-nonce="83865414718449(...)-9708f66daed2"
   data-tweet-stat-initialized="true"
   data-screen-name="Reuters"
   data-name="Reuters Top News" data-user-id="1652541"
   data-you-follow="true"
   data-follows-you="false"
   data-you-block="false"
   data-reply-to-users-json="[{<!-- JSON CONTENT>}]"
   data-disclosure-type=""
   data-has-cards="true">
   <!-- HTML CONTENT -->
 </div>
```

vagy öt friss tweet megjelenítéséhez:

```
<div class="new-tweets-bar js-new-tweets-bar"
     data-item-count="5">
  View 5 new Tweets
</div>
```

Ilyen esetekben csak a felhasználó által megadott Twitter-specifikus tanulóhalmaz alapján képes a mesterséges intelligencia besorolni az oldalelemeket az egyes kategóriákba.

OuterLink: Olyan hivatkozások, melyek egy külső webhelyre mutatva az aktuális webhelyről elviszik a crawlert. Ezek a linkek a crawlerek szempontjából azért bírnak jelentőséggel, mert kiválogatásukkal biztosítható egy webhely minél teljesebb végigjárása, majd annak végeztével az eltárolt külső hivatkozások valamelyikén folytathatja a crawlingot. Nehézség osztályozásukkor, hogy sok weblapon ugyanazon az átirányító oldalon keresztül hivatkozzák (pl. az index.hu a dex.hu-s linkekkel) a külső és belső linkeket, és ennek az átirányító lapnak a domain-je eltérhet a valódi webhelyétől. Ilyen esetben egy rossz megvalósítás hamisan sorolhat belső hivatkozásokat a külsők közé és viszont. Külső hivatkozás átirányítással az index.hu weblapon:

```
<a href="http://dex.hu/x.php?id=inxpfb&amp;
   url=http%3A%2F%2Fwww.portfolio.hu%2Fgazdasag%2F
   az_egesz_vilagot_pofon_vagna_a_fed_egy_ujabb_emelessel.
   245090.html">
    Az egész világot pofon vágná a Fed egy újabb emeléssel
</a>
```

InnerLink: Ezek a hivatkozások ugyanannak a webhelynek egy másik lapjára irányítanak. A kihívás az osztályozásukkor, hogy hasonlóan a külső oldalakra mutató hivatkozásokhoz, sok weblap átirányító oldalakon keresztül hivatkozik saját cikkeire, hogy így figyelhessék kereszthivatkozásaik forgalmát. Ezeket az oldalakat a különböző domain ellenére sem szabad az OuterPageLinkek közé sorolni. Az index.hu belső linkjei pl. így néznek ki:

```
<a href="http://dex.hu/x.php?id=inxcl&amp;
   url=http%3A%2F%2Findex.hu%2Ftudomany%2F2017%2F03%2F06%2F
   nem_periodikusan_csapodnak_a_foldre_az_aszteroidak%2F">
    Nem periodikusan csapódnak a Földre az aszteroidák
</a>
```

LanguageSelection: Ezek a hivatkozások az Euro News híroldalhoz hasonló többnyelvű oldalakon találhatók meg. Jelentőségük abban áll, hogy rájuk kattintva a jelenlegi oldalnak egy más nyelven íródott változatára jutunk. Ezeket a hivatkozások kiválogatva nagyméretű párhuzamos korpuszok építése válhat lehetővé. Osztályozásuk azért nehéz, mert eleinte csak azután lehetséges, hogy a hivatkozott oldalt letöltöttük. Egyedül a céloldal hasonlósága vagy az elem környezete különbözteti meg az egyszerű InnerLinkektől. Nyelvválasztó hivatkozás az Euro News honlapon:

```
<a href="http://hu.euronews.com/2017/03/06/merkel-nem-turi-ha
   -lenacizzak" lang="hu-HU" hreflang="hu">
    Magyar
</a>
```

9.2. A tanulóhalmaz

9.2.1. A tanulóhalmaz szerkezete

Az crawlinghoz használt mesterséges intelligencia működéséhez elengedhetetlen egy elégséges méretű tanulóhalmaz. Az implementáció egyik kihívása egy, ilyen tanulóhalmaz megépítésének felgyorsítására szolgáló eszköz elkészítése. Bár ez a feladat segédeszköz nélkül is elvégezhető, azonban a manuális munka mennyisége indokolja annak felgyorsítását. A böngészős környezetből adódóan azonban az implementáció nyelve eltér a dolgozat többi részétől, Python helyett JavaScript nyelvben kell megírni.

A második kihívás annak a lehetőségnek a biztosítása, hogy a „kibontott”, feldolgozott weboldalt tetszőleges adatkinyerő algorítmusokkal lehessen feldolgozni. Ehhez szükség volt egy megfelelő API biztosítására.

A harmadik a HTML kódrészletek átalakítása egy, a döntési fa által értelmezhető formátumba. Ez a formátum egy osztályként jelenik meg a kódban melynek tartalmaznia kell a HTML elem következő attribútumait:

tag : a HTML elem típusa (pl. *div, span, a*)

szabványos HTML attribútumok :

id, href és minden, a HTML szabványban meghatározott attribútum

nem szabványos HTML attribútumok :

pl. a 9.1-ban látható, HTML szabványok által nem felsorolt attribútumok.

forráskód : a HTML elem teljes forráskódja.

9.2.2. Tanulóhalmaz építése

A tanulóhamazt a halmazt a felhasználónak manuálisan kell összeállítania. A JSON formátumban tárolt tanulóhalmaz bejegyzései olyan adatstruktúrák, melyek tartalmazzák az elemek HTML HTML tagjét, kódját és a felhasználó által kiválasztott crawler osztályát. Az erősen repetitív munkavégzés meggyorsítása érdekében szükség volt egy eszközre, ami segít a felhasználónak közvetlenül a böngészőablakban kiválasztani és elmenteni a tanulóhalmaz elemei. Ez egy Chromium bővítmény formájában valósult meg, mely a következőképpen működik:

- A bővítmény az engedélyezés után minden kattintáskor megnyit egy felugró ablakot, ahol a felhasználó látja a kattintott elemet, illetve egy legördülő listából kiválaszthatja a crawler osztályt, amibe az elem tartozik.
- A bővítmény minden egyes elemet automatikusan feldolgoz, és a föntebb leírt tulajdonságait (tag, CSS class, crawler osztály) a megfelelő formátumban kiírja egy JSON fájlba.

A dolgozat részeként ilyen eszköz nem lett kidolgozva, a feladatra többnyire alkalmas Tolnai Gergő Chrome Element Sortere[30].

[30] `https://github.com/geritol/chrome-element-sorter`

9.3. A keretrendszer

A megvalósított program egy keretrendszer, mely képes több mesterséges intelligenciával (pl. Sci-Learn, Pandas) együttműködni. A szoftver az oldalt egy automatizált böngészővel tölti be, majd annak HTML-kódját a tanulóhalmazra rátanított mesterséges intelligenciával elemezteti. Az így osztályozott elemeket ezután osztályuk szerint vektorokba rakja (ld. 9.1). Az program által felismert elemosztályok sora bővíthető, a kezdeti verzióban azonban a következő osztályok kerültek bevezetésre:

Revealer : ebbe az osztályba kerülnek azok a HTML elemek, melyekre kattintva az oldalon több információ jelenik meg. Ilyen pl. a „Több komment mutatása", „Új bejegyzések megtekintése" és hasonló gombok.

Inner : ezek hagyományos hivatkozások, melyek a jelenlegi weboldalon belül mutatnak egy másik lapra.

Outer : szintén hagyományos hivatkozások, azonban az előzőekkel ellentétben ezek más weboldalakra hivatkoznak, „elvisznek" a jelenleg bányászott weboldalról.

Closer : ezek a gombok felelősek azért, hogy az oldalon felül előtűnő felugró blokkokat (pl. agresszvív reklámok, Twitteren megnyitott tweetek) bezárja, így lehetővé téve az eredeti oldal további feldolgozását.

A program először végigjárja a *Revealer* típusú linkeket. Mivel ezekből egy oldalon igen sok lehet, a felhasználó beállíthat egy felső határt a feldolgozott linkek mennyiségét illetően. Ezekkel párhuzamosan a program folyamatosan ellenőrzi a „Closer" típusú linkek jelenlétét az oldalon és ha ilyet talál, automatikusan aktiválja (rákattint). Ha ez a határ nincs beállítva, akkor az összes megtalált link feldolgozásra kerül, ez azonban hosszú időt és nagy számítókapacitást igényelhet. Miután ezek a linkek elfogytak a program rámegy egy véletlenszerű linkre. Itt egy felhasználó által megadott paraméter dönti el, hogy csak az aktuális weblapra mutató belső („Inner") vagy akár külső weblapokra mutató („Outer") hivatkozásokat is figyelembe vegye-e? Emellett beállítható, hogy hányszor maximum próbáljon meg görgetéssel új

adatokat előhívni a program, valamint, hogy a feldolgozás befejezte után elölről kezdje-e a folyamatot?

Ebbe a munkafolyamatba illeszthető be a tartalomkinyerés, mely a már feldolgozott vagy a még feldolgozás alatt álló oldalon futhat le. Ezt szintén a felhasználó állíthatja be annak tekintetében, hogy a revealer típusú linkek felugró típusú blokkokat jelenítenek meg, melyek utána bezáródnak, vagy a HTML oldalon permanensen új adatokat töltenek be.

IV. rész

Tesztek és eredmények

10. A tesztkörnyezet

A teszteket egy erősen korlátozott számítási kapacitással, Intel m3-as processzorral, 8Gb RAM-mal rendelkező laptopon, stabil 100Mb/s WiFi kapcsolattal végeztem el egy háttérben futó böngészővel és PyCharm IDE-vel.

A tesztet a következő oldalon végeztem el:
`https://twitter.com/searchdata_id=tweet%3A842698283604557824&f=tweets&vertical=default&q=Trump&src=tren`

Az implementáció után a keretrendszert megvizsgáltam sebesség és stabilitás szempontjából. A keretrendszer sebességét a következő beállításokkal mértem: A tesztelés során használt tartalomkinyerő a JusText volt. Ez a Twitter sok boilerplate-tel körülvett rövid szövegeit kinyerni nem tudta, de a kinyerés folyamata így is megtörtént, ezért sebességtesztelésre alkalmas volt. „CLOSED" típusú linkbejárást volt beállítva, tehát a program csak a belső linkek közül válogatott a Reveal típusú linkek feldolgozása után. Az *RTExtractor._extract_contly* True-ra volt állítva, tehát a tartalomkinyerő minden Reveal típusú elemre történt kattintás után lefutott, nem csak az egész oldal feldolgozása után egyszer. A legörgetések maximális számát 5-re, a feldolgozott Revealer típusú linkekét 10-re állítottam.

11. Teszteredmények

11.1. Sebesség

A teszteredmények [3] azt mutatták, hogy a folyamatos adatkinyerés (RTExtractor._extract_contly True értékre állítása) nagyjából 3 perccel hosszabbította meg az oldalfeldolgozás folyamatát, 13-ról 16 percre növelve. Az adatkinyerés teljes elhagyásával további 1 percet rövidült a futásidő.

3. táblázat. A sebesség tesztek eredménye

RTExtractor._extract_contly	tartalomkinyerő	Teszteredmény (sec)
True	Justext	956.974119926002
True	Justext	1038.443074727009
False	Justext	829.239110367998
False	Justext	751.222592491001
False	None	759.095069656992
False	None	664.459505462000

11.2. Stabilitás

Az első tesztek több, addig rejtett hibát feltártak a programban. Ezek kijavítása után a stabilitás javult, a tesztek hiba nélkül futottak le. Az internetkapcsolat megszakadása eredményezheti a folyamat megszakítását, ez azonban külső tényező.

11.3. Összehasonlítás más keresőrobotokkal

Összehasonlítva más, hagyományos weblapokra tervezett keresőrobotokkal az e dolgozatban bemutatott robot képes arra, hogy a tesztoldalon alapvetően rejtett, forráskód-szinten sem megjelenő oldalrészeket (pl. hozzászólások) megjelenítse és kinyerje. Míg az eddigi crawlerek csak az eredetileg betöltött odal forráskódját átvizsgálva tudták kinyerni a linkeket, illetve ezt a statikus kódtömeget tudták a tartalomkinyerők rendelkezésére bocsátani, addig ez a keretrendszer képessé teszi a tartalomkinyerőket az eddig bányászhatatlan oldalelemek vizsgálatára, valamint új bejárható külső-és belső linkeket tár fel a felhasználói interakció szimulálásával.

V. rész

Következtetések és jövőbeli munka

A fent felvázolt program a kívánt problémát képes lehet megoldani, ennek vizsgálatához azonban további tesztek és fejlesztések szükségesek. A moduláris felépítés miatt a rendelkezésre álló tanuló algoritmusok közül lehetséges kiválasztani a legmegfelelőbbet és azt alkalmazni. Ugyanez a megállapítás megáll a tartalomkinyerő technológiákra. A szoftver által feldolgozott oldal több információ kinyerését garantálhatja, mint a hagyományos megoldások, azonban az egyes kinyerőalgoritmusok hatékonyságának tesztelése dinamikus környezetben nem képezte e dolgozat tárgyát.

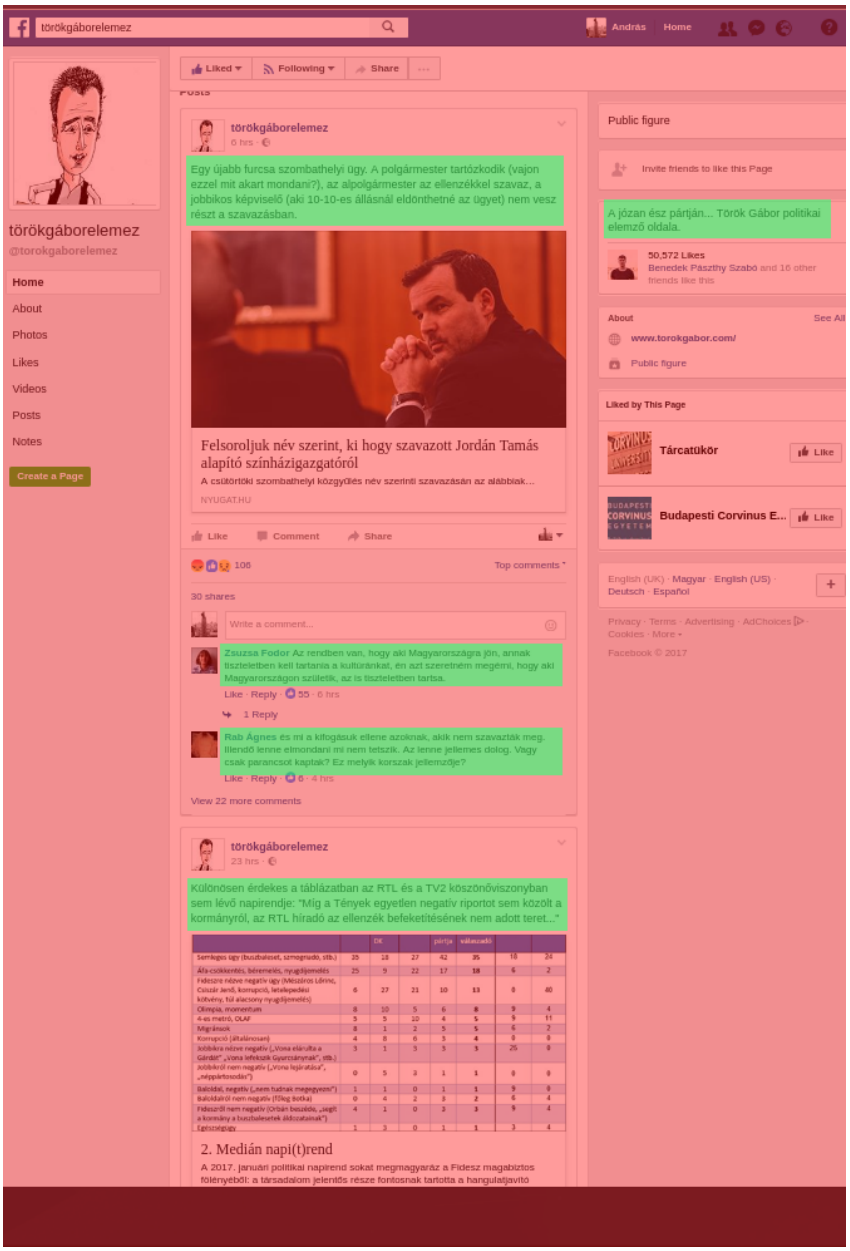
A módszer hátránya, hogy a böngészők, amiket az automatizáló rendszer használ erőforrásigényesek és sokszor lassabbak, mint a hagyományos keresőbotok. Előnye ugyanakkor, hogy e böngészők az erősen JavaScript-tel telített oldalakat is képes feldolgozni és, hogy nem oldalspecifikus (pl. Facebook Scraper), hanem általános megoldási javaslatot tesz a problémára.

A program fejlesztésének egyik fő iránya a sebesség növelése kell legyen. Bár a program működik, a hosszú futásidő kérdésessé teszi gyakorlati alkalmazhatóságát. A kutatás emellett tovább folytatható a legmegfelelőbb mesterséges intelligencia és kinyerőalgoritmus megkeresésével, hatékony headless, azaz rendermechanizmus nélkül működő böngészők integrálásával. Emellett megfontolandó a tartalomkinyerő és a keresőbot mesterséges intelligenciájának kombinálása, ami így egy lépésben tudná osztályozni az egyes oldalelemeket azok nyelvészeti és crawler-szempontú relevanciája szerint. Elképzelhető emellett még több osztály (pl. a cikkekben esetenként megjelenő lapozó linkek, vagy a szoftver mostani verziójában nem implementált nyelvválasztók) kezelése. Ezek halmaza az API-nak köszönhetően már most könnyen bővíthető.

Függelék

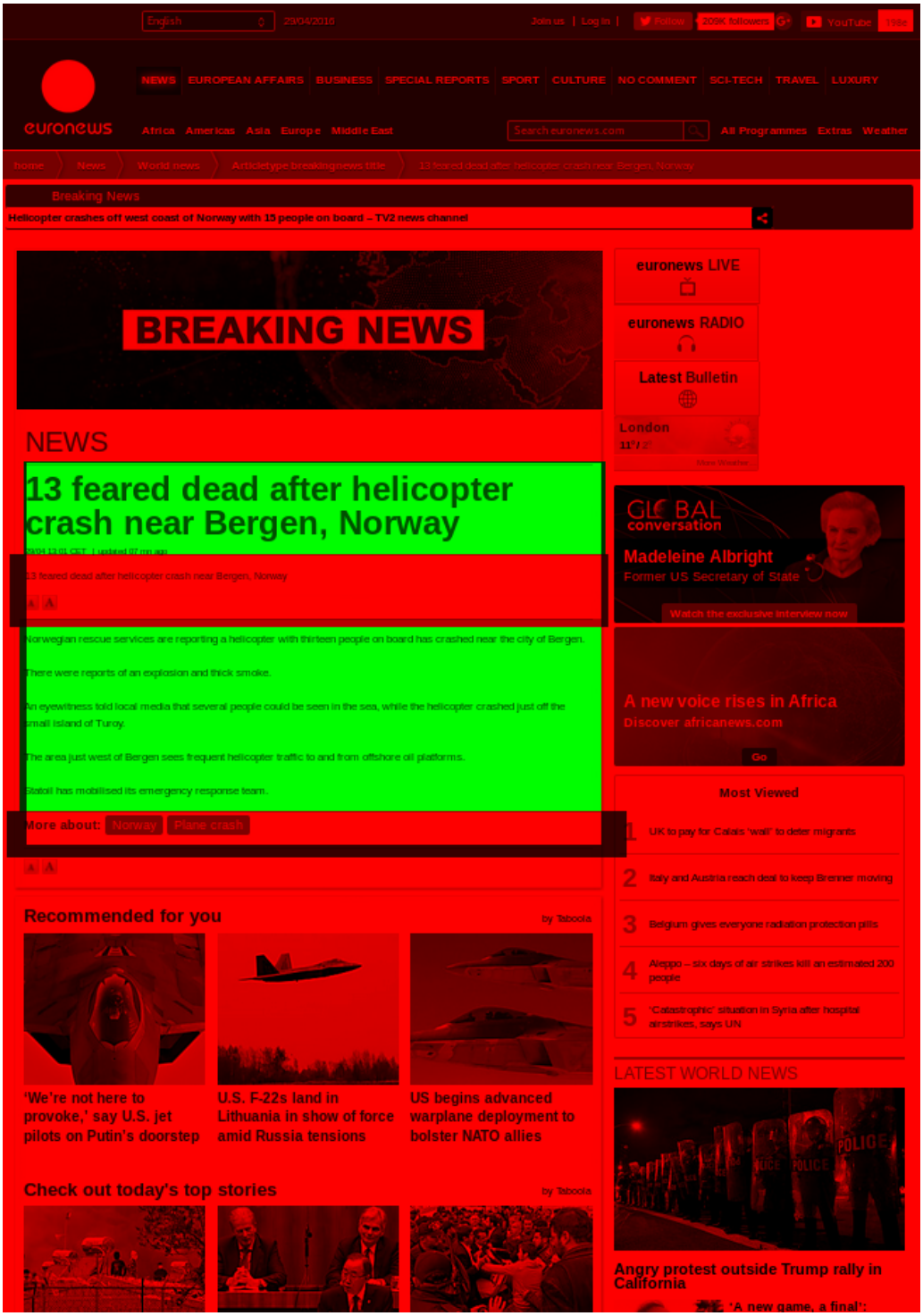
A. A boilerplate

A.1. Boilerplate és releváns tartalom az Facebookon

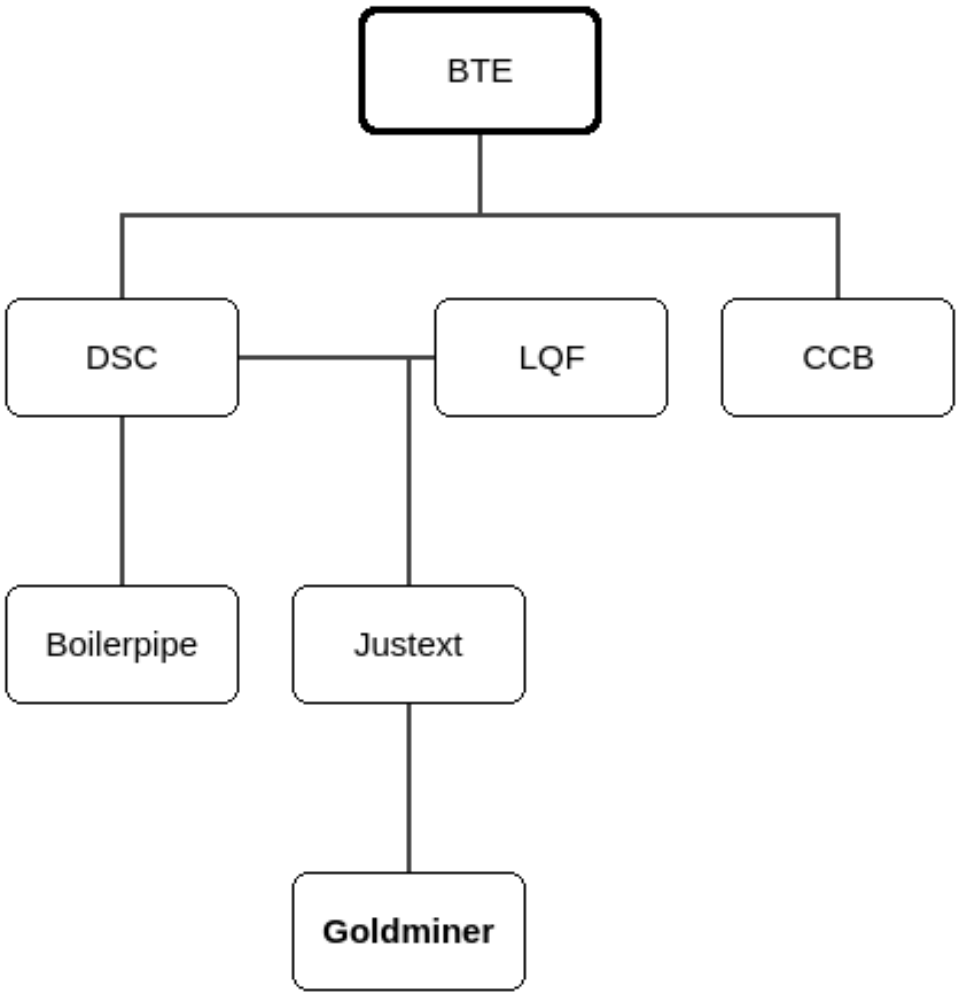


1. ábra. Sok oldalon nehéz megkülönböztetni.

A.2. Boilerplate és releváns tartalom az Euronews honlapján

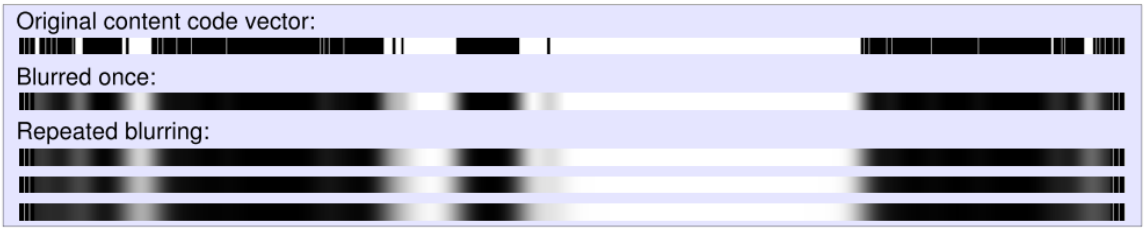


B. Kinyerőalgoritmusok



2. ábra. A kinyerőalgoritmusok származtatási ábrája.

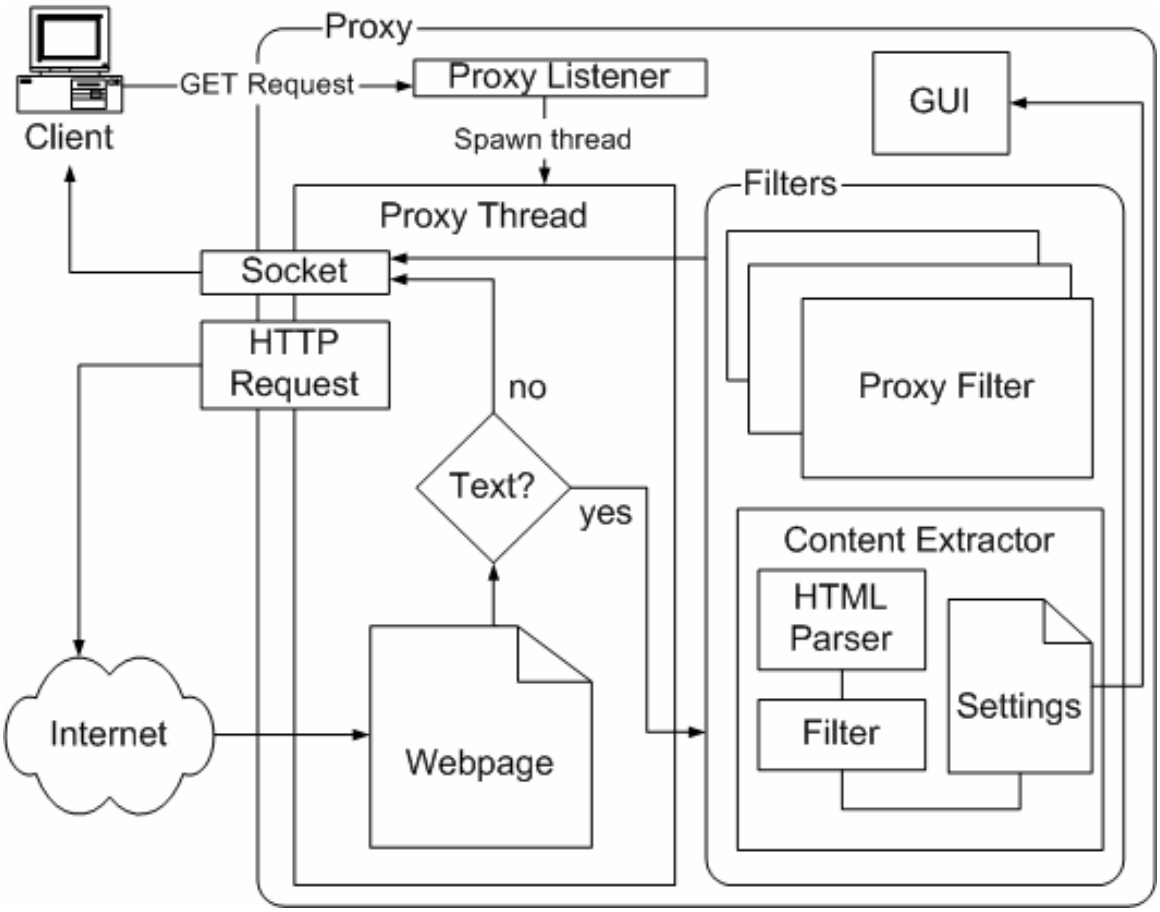
C. Content Code Blurring



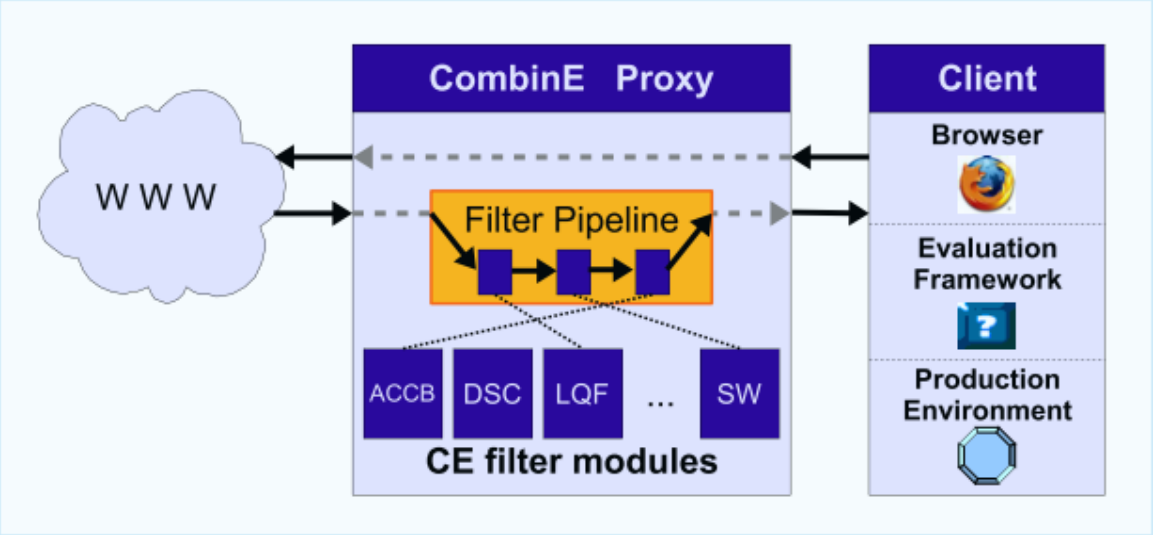
3. ábra. A Code Vector a CCB lefuttatása előtt és után szürkeárnyalatos képként [13].

D. A Crunch proxy

A Crunch proxy működési ábrája [3]

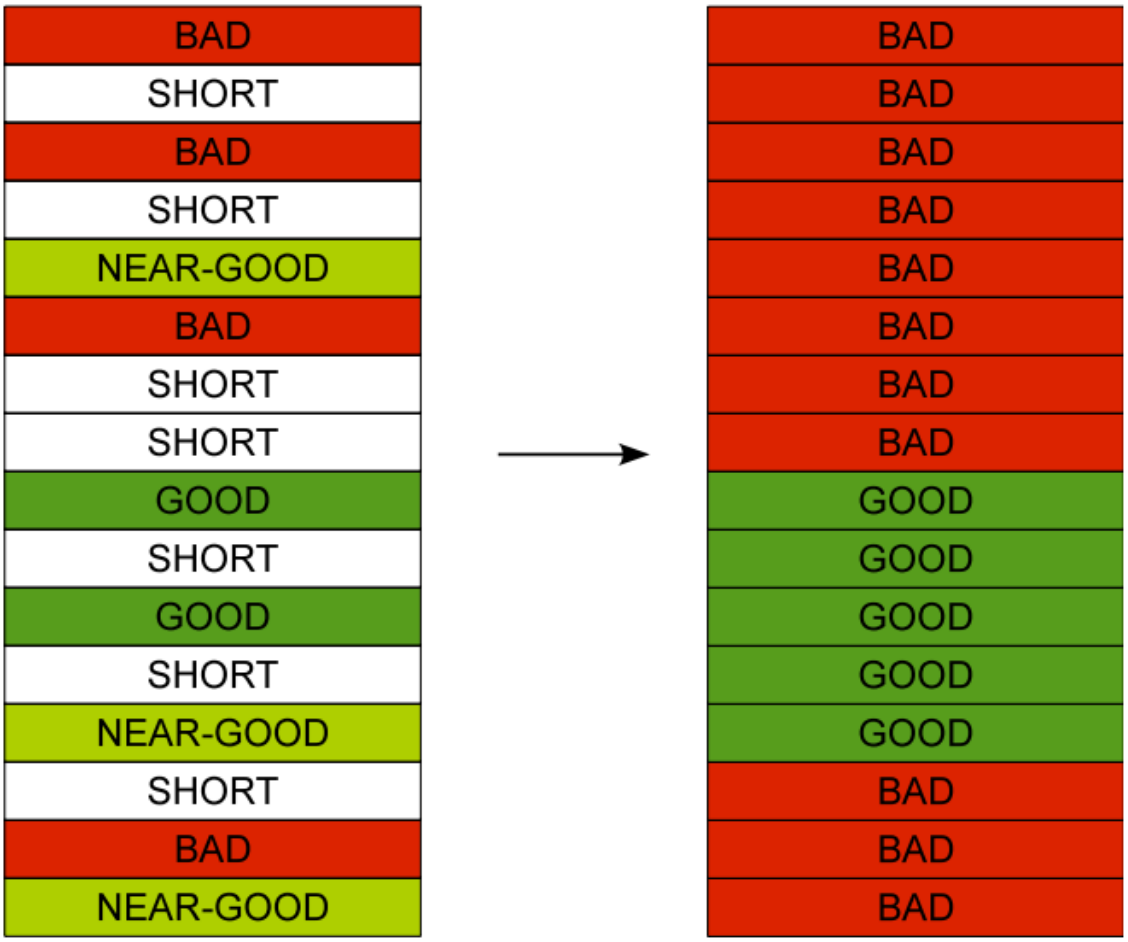


E. Combine System



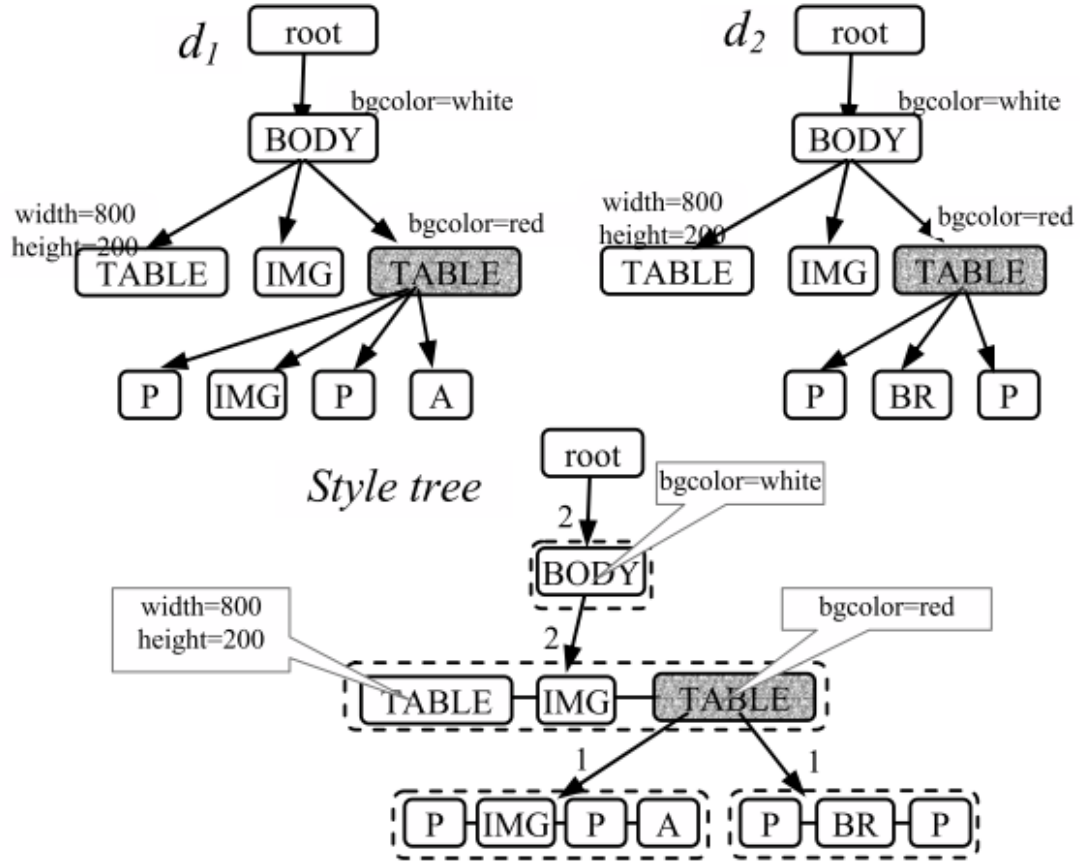
4. ábra. A CombinE System működési ábrája kaszkád működésnél [12]

F. JusText



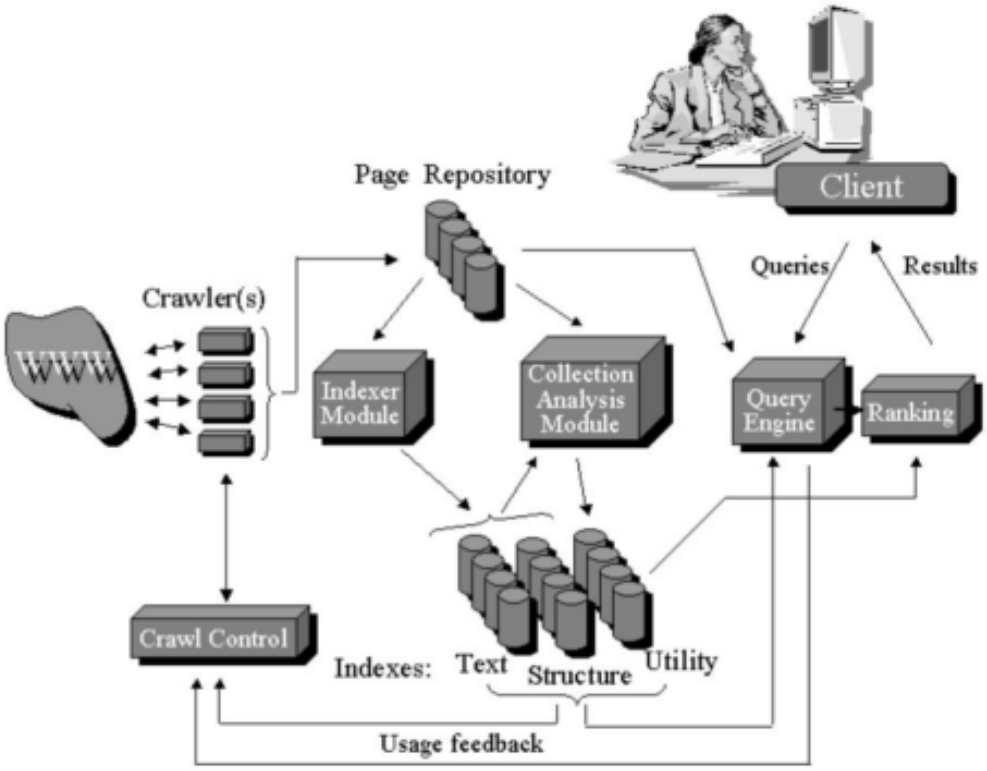
5. ábra. A JusText környezetfüggő osztályozása [12]

G. Site Style Tree



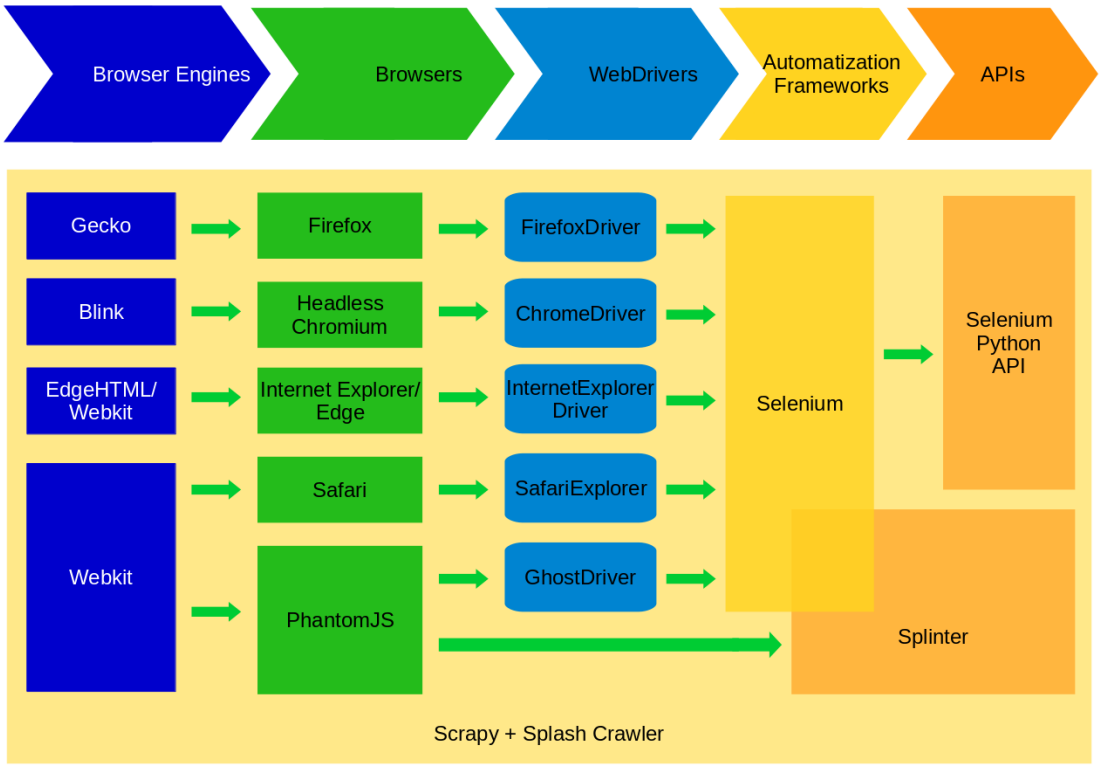
6. ábra. A Yi-féle Site Style Tree összeállítása két DOM-fából [24]

H. Keresőrobotok



7. ábra. Egy keresőrobot felépítése [27]

I. Böngészőautomatizálási Technológiák



8. ábra. A böngészőautomatizálási technológiák áttekintése

Hivatkozások

[1] J. Pomikalek, „Removing boilerplate and duplicate content from web corpora”, 2011.

[2] *The volume and evolution of web page template*, (2005. máj. 10.), ACM Press, 2005, old. 830–839, ISBN: 1-59593-051-5. DOI: 10.1145/1062745.1062763.

[3] S. Gupta, G. E. Kaiser, P. Grimm, M. F. Chiang, és J. Starren, „Automating content extraction of html documents”, 2004.

[4] E. Ferrara, P. De Meo, G. Fiumara, és R. Baumgartner, „Web data extraction, applications and techniques: A survey”, *Arxiv - Social Media Intelligence*, 70. évf., 301–323. old., 2014, ISSN: 09507051. DOI: 10.1016/j.knosys.2014.07.007. arXiv: 1207.0246. elérhető: http://dx.doi.org/10.1016/j.knosys.2014.07.007.

[5] R. T. Fielding. (2014). Hypertext transfer protocol (http/1.1): Semantics and content, elérhető: https://tools.ietf.org/html/rfc7231.

[6] W3Tech. (2015). Usage of content management systems for websites, elérhető: http://w3techs.com/technologies/overview/content_management/all.

[7] M. Nottingham és R. Sayre. (2005). The atom syndication format, elérhető: https://tools.ietf.org/html/rfc4287.

[8] S. McGlaun. (2012). Facebook data grows by over 500 tb daily, elérhető: https://www.slashgear.com/facebook-data-grows-by-over-500-tb-daily-23243691/ (elérés dátuma 2012. 08. 23.).

[9] P. Ast, M. Kapfenberger, és S. Hauswiesner, „Crawler Approaches And Technology”, *Graz University of Technology*, 1–19. old., 2008. elérhető: http://hyperg.iicm.edu/cguetl/courses/isr/uearchive/uews2008/Ue01%20-%20Crawler-Approaches-And-Technology.pdf.

[10] A. Finn, N. Kushmerick, és B. Smyth, „Fact or fiction: Content classification for digital libraries”, *Joint DELOS-NSF Workshop: Personalization and Recommender Systems in Digital Libraries*, 2001. DOI: 10.1.1.21.3834.

[11] D. Pinto, M. Branstein, R. Coleman, W. B. Croft, M. King, W. Li, és X. Wei, „QuASM: a system for question answering using semi-structured data", *Proceedings of the 2nd ACM/IEEE-CS joint conference on Digital libraries*, 46–55. old., 2002, ISSN: 1-58113-513-0. DOI: http://doi.acm.org/10.1145/544220.544228. elérhető: http://doi.acm.org/10.1145/544220.544228.

[12] T. Gottron, „Combining content extraction heuristics: The combine system", *Proceedings of the 10th International Conference on Information Integration and Web-based Applications and Services*, 2008, ISBN: 978-1-60558-349-5. DOI: 10.1145/1497308.1497418.

[13] ——, „Content code blurring: A new approach to content extraction", *Proceedings - International Workshop on Database and Expert Systems Applications, DEXA*, 29–33. old., 2008, ISSN: 15294188. DOI: 10.1109/DEXA.2008.43.

[14] S. Debnath és C. L. Mitra Prasenjitand Giles, „Automatic extraction of infor- mative blocks from webpages", *Special Track on Web Technologies and Applications, Foundations of Intelligent Systems*, (2005. máj. 28.), 2009, ISBN: 978-3-540-25878-0.

[15] C. Mantratzis, M. Orgun, és S. Cassidy, „Separating xhtml content from navigation clutter using dom-structure block analysis", *Proceeding WWW '05, Proceedings of the six- teenth ACM conference on Hypertext and hypermedia*, (2005), ACM Press, 2005, 145–147. old.

[16] S. Louvan, „Extracting the main content from html documents", 2009.

[17] T. Kovačič, „Evaluating web content extraction algorithms", 2012.

[18] S. Gupta, G. E. Kaiser, P. Grimm, M. F. Chiang, és J. Starren, „Automating content extraction of html documents", 2004.

[19] Y. Weissig és T. Gottron, „Combination of content extraction algorithms", 2004.

[20] C. Kohlschütter, P. Fankhauser, és W. Nejdl, „Boilerplate Detection using Shallow Text Features", *Text*, 441–450. old., 2010, ISSN: 160558889X. DOI: 10.1145/1718487.1718542. elérhető: http://portal.acm.org/citation.cfm?id=1718542.

[21] M. Spousta, M. Marek, és P. Pecina, „Victor: the Web-Page Cleaning Tool”, *Workshop Programme*, 12. old., 2007. elérhető: http://www.lrec-conf.org/proceedings/lrec2008/workshops/W19%7B%5C_%7DProceedings.pdf%7B%5C#%7Dpage=18.

[22] Z. Bar Yossef és S. Rajagopalan, „Template detection via data mining and its applications”, *WWW '02 Proceedings of the 11th international conference on World Wide Web*, 2002, ISBN: 1-58113-449-5. DOI: 10.1145/511446.511522.

[23] a. Broder, „Identifying and filtering near-duplicate documents”, *Combinatorial Pattern Matching*, 1–10. old., 2000, ISSN: 16113349. DOI: 10.1007/3-540-45123-4_1. elérhető: http://www.springerlink.com/index/KTN21YJUL3R379XY.pdf.

[24] L. Yi, B. Liu, és X. Li, „Eliminating noisy information in Web pages for data mining”, *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '03*, 296. old., 2003. DOI: 10.1145/956755.956785. elérhető: http://portal.acm.org/citation.cfm?doid=956750.956785.

[25] D. Gibson, K. Punera, és A. Tomkins, „The volume and evolution of web page templates”, *Special interest tracks and posters of the 14th international conference on World Wide Web*, 830–839. old., 2005. DOI: http://doi.acm.org/10.1145/1062745.1062763. elérhető: http://doi.acm.org/10.1145/1062745.1062763.

[26] I. Endrédy és A. Novak, „More effective boilerplate removal-the goldminer algorithm”, *Polibits*, 48. sz., 79–83. old., 2013.

[27] A. Arasu, J. Cho, H. Garcia-Molina, A. Paepcke, és S. Raghavan, „Searching the web”, *ACM Trans. Internet Technol.*, 1. évf. 1. sz., 2–43. old., 2001. aug., ISSN: 1533-5399. DOI: 10.1145/383034.383035. elérhető: http://doi.acm.org/10.1145/383034.383035.