Nejc Povšič, Vid Ribič, Luka Bezovšek

1. seminar pri predmetu Iskanje in ekstrakcija podatkov s spleta

MENTORJA: prof. dr. Marko Bajec in doc. dr. Slavko Žitnik

Na spletu danes lahko najdemo neskončno mnogo podatkov. Zahvaljujoč iskalnikom (npr. Google), lahko v množici vsebin najdemo tiste, ki nas resnično zanimajo. Ker tovrstna orodja vseh strani ne morejo pregledati v doslednem času, v ozadju ves čas tečejo posebni programi, ki iz te množice podatkov avtomatsko pridobivajo nove/sveže podatke, po katerih nato iščemo uporabniki. V sklopu naloge si bomo pogledali primer implementacije tovrstnega spletnega pajka.

1. Uvod

Za prvi seminar smo imeli nalogo ustvariti spletnega pajka za spletne strani v domeni *gov.si*.

2. Realizacija spletnega pajka

V tem razdelku bomo opisali potek razvoja in implementacijo našega pajka ter se dotaknili problemov in rešitev, ki smo jih uporabili.

2.1. Izbira programskega jezika

Prva odločitev, ki smo jo morali sprejeti, preden smo sploh lahko začeli z razvojem, je bila, kateri programski jezik bomo uporabljali. Po kratki razpravi smo se odločili za Python, s katerim smo vsi dovolj dobro seznanjeni, kar nam je omogočilo, da smo se lahko posvetili sami implementaciji pajka, ne pa specifikam jezika.
Python ima na voljo tudi veliko knjižnic, ki se uporabljajo za spletno iskanje in luščenje podatkov.

2.2. Kako smo implementirali več workerjev?

Naš pajek uporablja več ločenih procesov, ki pridobivajo strani in iz njih luščijo podatke. Uporabljamo Python knjižnico *multiprocessing*, s katero lahko ustvarjamo ločene procese z objektom *Process*. Procesi komunicirajo s podatkovno bazo, kjer je realiziran tudi frontier. Zaradi večih procesov, ki pridobivajo stran iz frontierja, bi lahko kdaj prišlo do situacije, kjer dva ločena procesa dobita isto stran iz frontierja. Ta problem smo rešili z uporabo objekta *Lock* iz knjižnice *multiprocessing*.
Lock deluje tako, da v nekem trenutku “zaklene” funkcijo, da lahko do nje dostopa samo en proces, ostali pa čakajo na sprostitev ključavnice. V razredu, ki je ustvaril procese, smo ustvarili nov objekt tipa *Lock* in ga preko argumentov prenesli na vse procese, saj mora za pravilno delovanje vsak proces uporabljati isto ključavnico. *Lock* je bil nato uporabljen pri pridobivanju strani iz frontierja. Ko je nek proces začel z izvajanjem funkcije, jo je zaklenil, ko

pa se je funkcija izvedla do konca, je ključavnico sprostil. Primer delovanje lahko vidimo na spodnjih slikah.

```
class Crawler:
    def __init__(self, number_of_processes):
        self.number_of_processes = number_of_processes

        self.lock = Lock()

        with open("seed_pages.txt", "r") as seed_pages:
            for seed_page in seed_pages:
                if "#" not in seed_page:
                    database_handler.add_seed_page_to_frontier(seed_page.strip())

        # When starting the crawler reset frontier active flags
        database_handler.reset_frontier()

    def run(self):
        for i in range(self.number_of_processes):
            p = Process(target=self.create_process, args=[i, self.lock])
            p.start()

    def create_process(self, index, lock):
        crawler_process = CrawlerProcess(index, lock)
```

V razredu *Crawler* ustvarimo *Lock*, ki ga nato dovedemo do procesov

```
"""
    current_page is a dictionary with an id (database id for updating) and url field
"""
self.current_page = database_handler.get_page_from_frontier(self.lock)
```

V procesu *CrawlerProcess* se kliče funkcija *get_page_from_frontier*, ki pridobi stran iz frontierja

```
    """
        This function uses the lock so that no two crawler processes get the same frontier url
    """

    def get_page_from_frontier(self, lock):
        lock.acquire()
```

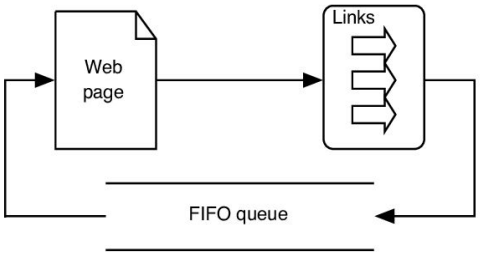
Vsakič, ki se funkcija pokliče, se ključavnica zaklene, kar prepreči ostalim procesom, da v tistem času dostopajo do frontierja

```
    finally:
        if connection:
            self.connection_pool.putconn(connection)

        lock.release()
```

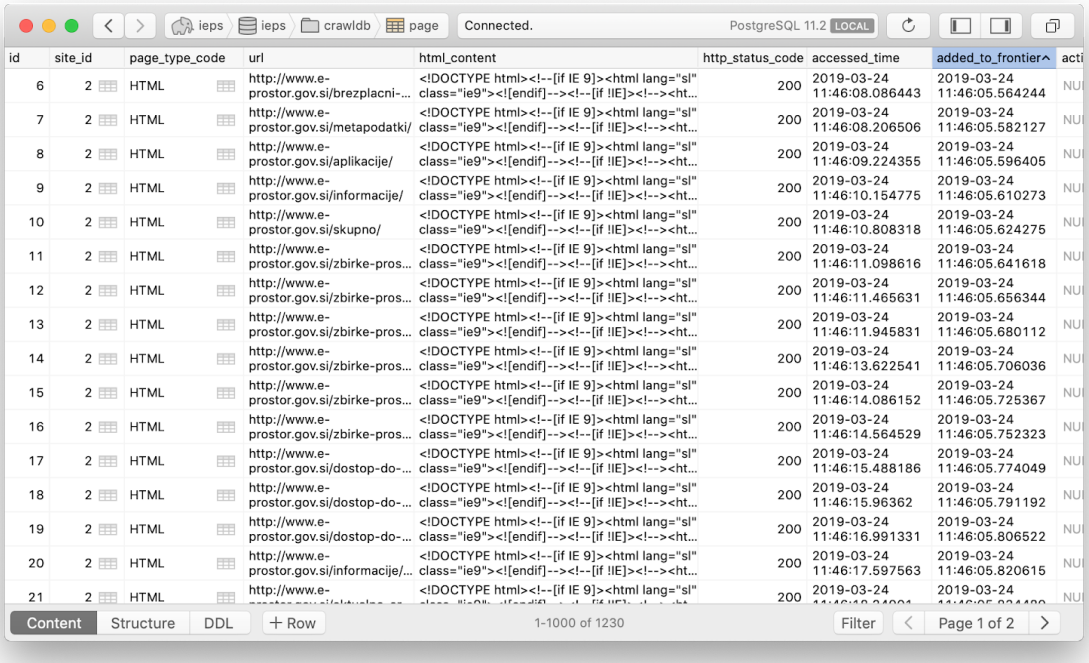
Ko se funkcija izvede do konca, proces ključavnico sprosti

2.3. Kako smo realizirali strategijo breadth-first?

Pajek, ki uporablja strategijo breadth-first, išče v širino. Če bi iskal v globino, bi spletno stran, ki jo je vzel iz frontiera, nadomestil z novimi spletnimi stranmi, ki jih je našel. Pri iskanju v širino pa jih doda na konec vrste. Frontier je v tem primeru tako realiziran kot FIFO vrsta (angl. first in, first out). Vsak naslednji URL vpišemo na konec seznama, ko beremo iz frontiera, pa vzamemo prvega, ki je v vrsti. Izkaže se, da pri iskanju v širino zajamemo bolj raznolike vsebine.



FIFO vrsto smo realizirali tako, da smo tabeli *page* dodali stolpec *added_at_time*, v katerega smo zapisali natančen čas, ko je bila spletna strana dodana v frontier. Pri branju iz frontiera smo potem rezultate iz baze uredili časovno padajoče glede na ta stolpec. Tako smo dobili vrsto, v kateri so bile na prvem mestu spletne strani, ki so bile v bazo dane prej.



ieps › ieps › crawldb › page — Connected. — PostgreSQL 11.2 LOCAL

id	site_id	page_type_code	url	html_content	http_status_code	accessed_time	added_to_frontier	acti
6	2	HTML	http://www.e-prostor.gov.si/brezplacni-...	<!DOCTYPE html><!--[if IE 9]><html lang="sl" class="ie9"><![endif]--><!--[if !IE]><!--><ht...	200	2019-03-24 11:46:08.086443	2019-03-24 11:46:05.564244	NU
7	2	HTML	http://www.e-prostor.gov.si/metapodatki/	<!DOCTYPE html><!--[if IE 9]><html lang="sl" class="ie9"><![endif]--><!--[if !IE]><!--><ht...	200	2019-03-24 11:46:08.206506	2019-03-24 11:46:05.582127	NU
8	2	HTML	http://www.e-prostor.gov.si/aplikacije/	<!DOCTYPE html><!--[if IE 9]><html lang="sl" class="ie9"><![endif]--><!--[if !IE]><!--><ht...	200	2019-03-24 11:46:09.224355	2019-03-24 11:46:05.596405	NU
9	2	HTML	http://www.e-prostor.gov.si/informacije/	<!DOCTYPE html><!--[if IE 9]><html lang="sl" class="ie9"><![endif]--><!--[if !IE]><!--><ht...	200	2019-03-24 11:46:10.154775	2019-03-24 11:46:05.610273	NU
10	2	HTML	http://www.e-prostor.gov.si/skupno/	<!DOCTYPE html><!--[if IE 9]><html lang="sl" class="ie9"><![endif]--><!--[if !IE]><!--><ht...	200	2019-03-24 11:46:10.808318	2019-03-24 11:46:05.624275	NU
11	2	HTML	http://www.e-prostor.gov.si/zbirke-pros...	<!DOCTYPE html><!--[if IE 9]><html lang="sl" class="ie9"><![endif]--><!--[if !IE]><!--><ht...	200	2019-03-24 11:46:11.098616	2019-03-24 11:46:05.641618	NU
12	2	HTML	http://www.e-prostor.gov.si/zbirke-pros...	<!DOCTYPE html><!--[if IE 9]><html lang="sl" class="ie9"><![endif]--><!--[if !IE]><!--><ht...	200	2019-03-24 11:46:11.465631	2019-03-24 11:46:05.656344	NU
13	2	HTML	http://www.e-prostor.gov.si/zbirke-pros...	<!DOCTYPE html><!--[if IE 9]><html lang="sl" class="ie9"><![endif]--><!--[if !IE]><!--><ht...	200	2019-03-24 11:46:11.945831	2019-03-24 11:46:05.680112	NU
14	2	HTML	http://www.e-prostor.gov.si/zbirke-pros...	<!DOCTYPE html><!--[if IE 9]><html lang="sl" class="ie9"><![endif]--><!--[if !IE]><!--><ht...	200	2019-03-24 11:46:13.622541	2019-03-24 11:46:05.706036	NU
15	2	HTML	http://www.e-prostor.gov.si/zbirke-pros...	<!DOCTYPE html><!--[if IE 9]><html lang="sl" class="ie9"><![endif]--><!--[if !IE]><!--><ht...	200	2019-03-24 11:46:14.086152	2019-03-24 11:46:05.725367	NU
16	2	HTML	http://www.e-prostor.gov.si/zbirke-pros...	<!DOCTYPE html><!--[if IE 9]><html lang="sl" class="ie9"><![endif]--><!--[if !IE]><!--><ht...	200	2019-03-24 11:46:14.564529	2019-03-24 11:46:05.752323	NU
17	2	HTML	http://www.e-prostor.gov.si/dostop-do-...	<!DOCTYPE html><!--[if IE 9]><html lang="sl" class="ie9"><![endif]--><!--[if !IE]><!--><ht...	200	2019-03-24 11:46:15.488186	2019-03-24 11:46:05.774049	NU
18	2	HTML	http://www.e-prostor.gov.si/dostop-do-...	<!DOCTYPE html><!--[if IE 9]><html lang="sl" class="ie9"><![endif]--><!--[if !IE]><!--><ht...	200	2019-03-24 11:46:15.96362	2019-03-24 11:46:05.791192	NU
19	2	HTML	http://www.e-prostor.gov.si/dostop-do-...	<!DOCTYPE html><!--[if IE 9]><html lang="sl" class="ie9"><![endif]--><!--[if !IE]><!--><ht...	200	2019-03-24 11:46:16.991331	2019-03-24 11:46:05.806522	NU
20	2	HTML	http://www.e-prostor.gov.si/informacije/...	<!DOCTYPE html><!--[if IE 9]><html lang="sl" class="ie9"><![endif]--><!--[if !IE]><!--><ht...	200	2019-03-24 11:46:17.597563	2019-03-24 11:46:05.820615	NU
21	2	HTML	http://www.e-	<!DOCTYPE html><!--[if IE 9]><html lang="sl"	200	2019-03-24	2019-03-24	NU

Content | Structure | DDL | + Row — 1-1000 of 1230 — Filter — Page 1 of 2

2.4. Kako se pajek izogiba pastem?

Pasti na katere lahko naleti pajek so URL naslovi, katerih dolžina se s časom povečuje (v vsaki iteraciji se v frontier doda nov, daljši URL naslov z novimi podstranmi). Ker se v tem primeru pajek lahko ujame v neskončno zanko, smo tovrstni problem rešili z omejevanjem dolžine URL naslovov. Dolžino naslova smo navzgor omejili z 2000 znaki. V primeru, da bi

kateri izmed URL naslovov strani, ki bi se morala uvrstiti v frontier, presegel zgornjo mejo, bi se ta stran in posledično tudi vse njene podstrani zavrgla.

```
def add_seed_page_to_frontier(self, seed_page):
    connection = None

    if len(seed_page) <= MAX_URL_LEN:
        try:
            connection = self.connection_pool.getconn()

            cursor = connection.cursor()

            cursor.execute(
                """
                    INSERT INTO crawldb.page("url", "page_type_code", "added_at_time")
                    VALUES(%s, %s, %s);
                """,
                (seed_page, "FRONTIER", datetime.now())
            )

            connection.commit()

            cursor.close()
        except (Exception, psycopg2.DatabaseError) as error:
            print("[ERROR WHILE ADDING PAGE TO FRONTIER]", error)
        finally:
            if connection:
                self.connection_pool.putconn(connection)
```

Dodajanje strani v frontier in preverjanje dolžine URL naslova z namenom izogibanja pastem

2.5. Zaznavanje dupliciranih spletnih strani

Za potrebe zaznavanja dupliciranih spletnih strani smo implementirali dve metodi, ki sta osnovani na analizi vsebine strani. S prvo metodo najprej preverimo, ali je nova stran popolnoma identična kateri izmed strani, ki jih je pajek že prebral. To storimo tako, da celotno vsebino strani zgostimo s funkcijo, ki nam za določeno vsebino vedno vrne isti niz znakov - v našem primeru smo uporabili zgoščevalno funkcijo SHA-256. Ker za vsako preteklo stran v podatkovni bazi hranimo vrednost zgoščevanja, lahko z enostavno SQL poizvedbo najdemo morebitno ujemanje med zgoščeno vrednostjo trenutne in zgoščenimi vrednostmi preteklih strani. V primeru, da smo našli popolno ujemanje, trenutno stran lahko označimo kot duplikat.

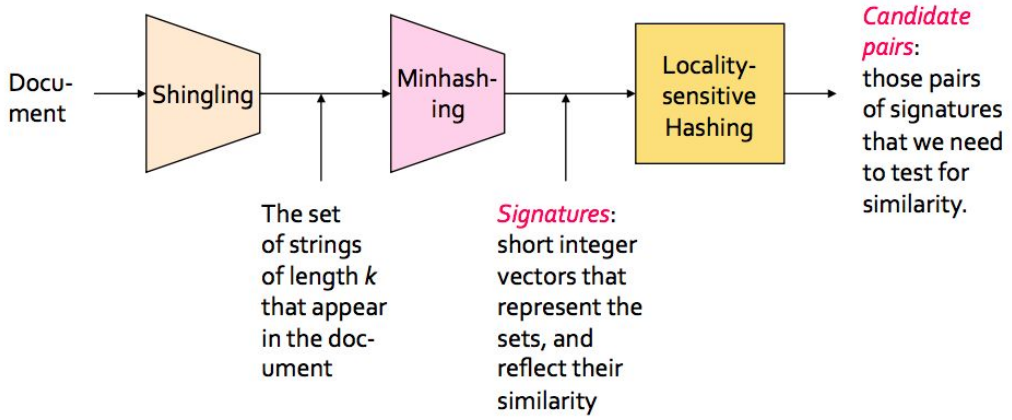
```python
def create_content_hash(self, html_content):
    try:
        m = hashlib.sha256()

        m.update(html_content.encode('utf-8'))

        return m.hexdigest()
    except Exception as error:
        print("     [CRAWLING] Error while creating content hash", error)

        return None
```

Zgoščevanje vsebine strani

```python
"""
    Find a page in the database by the hash_content and return it if it exists
"""

def find_page_duplicate(self, hash_content):
    connection = None

    try:
        connection = self.connection_pool.getconn()

        cursor = connection.cursor()

        cursor.execute(
            """
                SELECT * FROM crawldb.page WHERE hash_content=%s;
            """,
            (hash_content,)
        )

        connection.commit()

        page = cursor.fetchone()

        return page is not None

    except (Exception, psycopg2.DatabaseError) as error:
        print("[ERROR WHILE CHECKING PAGE DUPLICATE]", error)
    finally:
        if connection:
            self.connection_pool.putconn(connection)
```

Iskanje obstoječih strani z enakim izvlečkom

Zgoraj opisana metoda stran prepozna kot duplikat samo v primeru, ko gre za popolnoma identično stran kateri izmed že pregledanih strani. V primeru, ko je stran zelo podobna kateri izmed preteklih (npr. spremenjenih je samo nekaj besed/stavkov), pa ta metoda ne deluje, saj je rezultat zgoščevanja vsebine drugačen. Zato smo implementirali še drugo metodo, pri kateri smo uporabili postopek imenovan “Locality-Sensitive-Hashing”, s pomočjo katerega lahko na relativno enostaven način določimo stopnjo ujemanja dveh strani.



Preden se lotimo samega iskanja podobnih strani, moramo vsebino posamezne strani ustrezno preobdelati. Če gledamo na vsebino strani kot na besedilo oz. dokument spletne strani, ta vsebuje tudi HTML značke, katerih nabor je relativno kratek, veliko izmed njih pa je skupnih večini strani. Da se izognemo primerjanju vsebine na podlagi rezerviranih HTML značk, te enostavno odstranimo iz besedila ter ohranimo samo vidni del vsebine strani. Dokument, ki ga razčlenjujemo, razdelimo na več podenot (angl. shingles), poljubne dolžine *k* (angl. k-shingles). Recimo, če želimo stavek *"Danes je sončno vreme, jutri bo dež."* razdeliti na podenote dolžine k=5, dobimo seznam nizov *"Danes"*, *"anes_"*, *"nes_j"*, *"es_je"* itn. Pridobljene shingle nato zgostimo (angl. hash) z neko zgoščevalno funkcijo h, da dobimo seznam števil z enako dolžino. V našem primeru smo dolžino podenot nastavili na 10 znakov, ki jih nato zgostimo v števila s fiksno dolžino 32 bitov.

```
    """
        Split  text to shingles of size SHINGLE_SIZE and output them as integers of fixed size
    """
    def text_to_shingle_set(self, text):
        words = text.split()

        # keeps word shingles
        shingles_in_doc_words = set()

        # keeps hashed shingles
        shingles_in_doc_ints = set()

        shingle = []

        shingle_size = SHINGLE_SIZE

        for index in range(len(words) - shingle_size + 1):
            shingle = words[index:index + shingle_size]
            shingle = ' '.join(shingle)

            # Hash the shingle to a 32-bit integer (create token)
            crc = binascii.crc32(shingle.encode()) & 0xffffffff

            if shingle not in shingles_in_doc_words:
                shingles_in_doc_words.add(shingle)

            if crc not in shingles_in_doc_ints:
                shingles_in_doc_ints.add(crc)

        return shingles_in_doc_ints
```

Implementacija razdelitve besedila na podenote in zgoščevanje podenot - rezultat je seznam števil dolžine 32b

Seznam podenot oz. zgoščenih nizov si lahko za vsako stran predstavljamo kot množico s končnim številom elementov. Podobnost posameznih strani lahko sedaj določimo s pomočjo Jaccardove razdalje, ki je definirana kot količnik med presekom in unijo dveh množic, oz. razmerje med številom skupnih elementov in številom vseh elementov v obeh množicah. Sklepamo, da bodo dokumenti, ki so si med seboj bolj podobni, imeli več skupnih elementov (shinglov) kot tisti, ki si niso. Prav tako lahko predpostavimo, da bolj kot sta si dokumenta *d1* in *d2* podobna, bolj sta si podobna tudi njuna izvlečka.

$$J(A,\ B)\ =\ \frac{(A \cap B)}{(A \cup B)}$$

Iskanje skupnih elementov (shinglov) med trenutno stranjo in že pregledanimi stranmi smo implementirali s pomočjo SQL poizvedbe, ki vrne število skupnih elementov posamezne strani in jih uredi po padajočem vrstnem redu.

```
"""
    Check out what is the percentage of similarity between current page containing set of hash signatures and
    already crawled pages
"""

def calculate_biggest_similarity(self, signatures):
    connection = None

    try:
        connection = self.connection_pool.getconn()

        cursor = connection.cursor()

        cursor.execute(
            """
                SELECT *
                FROM   crawldb.content_hash i, LATERAL (
                    SELECT count(*) AS ct
                    FROM   unnest(i.hash) signature
                    WHERE  signature = ANY(%s::bigint[])
                ) x, LATERAL (
                    SELECT (i.hash_length + %s) AS total_sum
                ) y
                ORDER  BY x.ct DESC, hash_length ASC;
            """,
            (str(signatures), len(signatures))
        )

        connection.commit()

        first_matching = cursor.fetchone()

        if first_matching is None:
            return 0
```

```
    except (Exception, psycopg2.DatabaseError) as error:
        print("[ERROR WHILE FINDING SIMILAR PAGES]", error)
    finally:

        if connection:
            self.connection_pool.putconn(connection)

    if first_matching[4] > 0:

        # calculate jaccard similarity (intersection over union)
        return first_matching[4] / (first_matching[5] - first_matching[4])

    else:
        return 0
```

Izračun podobnosti med stranmi z uporabo metode LSH in Jaccardove razdalje

Opisana metoda nam tako vrne rezultat na območju med 0 in 1, pri čemer 0 pomeni, da sta dve strani popolnoma različni, 1 pa pomeni, da si strani popolnoma enaki. Za tretiranje dveh strani kot identični, smo določili spodnjo mejo 0.95, kar pomeni, da smo v primeru, ko je metoda vrnila rezultat večji od te meje, stran označili kot duplikat.

Primera strani, ki jih je program zaznal kot duplikata sta strani:

- https://e-uprava.gov.si/podrocja/drzava-druzba.html?caps_mode=0
- https://e-uprava.gov.si/podrocja/drzava-druzba.html?caps_mode=1.

Če si pogledamo vsebino strani vidimo, da se vsebina strani razlikuje le v velikosti zapisa besedila. V primeru, ko ima parameter caps_mode vrednost 0, besedilo strani zapisano z malimi tiskanimi črkami in z velikimi tiskanimi črkami, ko je njegova vrednost enaka 1. Ker je bila v preteklosti stran, ki v naslovu ne vsebuje parametra caps_mode, njena vsebina pa je enaka zgornjima stranema, že prebrana, sta bili drugi dve strani prepoznani kot duplikata.

```
[CRAWLING] Found page duplicate, that has already been parsed:  https://e-uprava.gov.si/podrocja/drzava-druzba.html?caps_mode=1
[CRAWLING] Found page duplicate, that has already been parsed:  https://e-uprava.gov.si/podrocja/drzava-druzba.html?bold_mode=0
```

Zamenjava besed v odstavku nima večjega vpliva na shingle, zato lahko s tem pristopom kot duplikate zaznavamo tudi strani, ki sicer nimajo povsem identične vsebine, so pa si med seboj zelo podobne.

2.6. Delovanje razčlenjevalnika (angl. parser)

HTML vsebino spletne strani pridobimo z uporabo Selenium knjižnice in Headless Chrome. Ta omogoča, da spletno stran upodobimo tako, kot bi se upodobila v brskalniku, saj zažene tudi Javascript kodo. Ta se uporablja predvsem za ustvarjanje dinamičnih spletnih strani, ki se postavijo šele po izvedbi. Pridobivanje samega html dokumenta brez izvedbe Javascripta s strežnika je zato pogostokrat nepopolno, saj manjkajo ključni elementi spletne strani.
Za razčlenjevalnik smo sprva uporabili kar knjižnico BeautifulSoup, ki iz HTML besedila ustvari drevesno strukturo, iz katere lahko nato pridobimo elemente in njihove atribute. Imeli

smo nalogo pridobiti oznake <a> in <img>, iz katerih smo nato morali izluščiti atribute *href* (iz <a>) in *src* (iz <img>).
Z uporabo BeautifulSoup smo pridobili vse potrebno, ampak smo naleteli na problem relativnih URL-jev. BeautifulSoup zna prebrati samo besedilo iz HTML oznak, kar pomeni, da vse atribute pridobi tako, kot so zapisani v oznakah. To pomeni, da relativni URL-ji ostanejo relativni, ko jih pridobimo. Problem smo na začetku rešili tako, da smo vsako povezavo pregledali in dodali domeno, če je bil URL relativen. A kmalu smo opazili, da to ni dovolj, saj se je kdaj zgodilo, da domena ni bila osnovni URL (ang. “base URL”), iz katerega so se nato zgradili absolutni URL-ji. Na spodnji sliki lahko vidimo razliko med atributom, ki je viden v HTML-ju, in povezavo, ki jo dejansko vidi spletni brskalnik za element <a> na strani *http://evem.gov.si/evem/drzavljani/zacetna.evem*.

```
anchor.getAttribute('href')
"drzavljani/zacetna.evem"
anchor.baseURI
"http://evem.gov.si/evem/"
document.baseURI
"http://evem.gov.si/evem/"
anchor.href
"http://evem.gov.si/evem/drzavljani/zacetna.evem"
```

Kot vidimo, za to stran dobimo domeno *http://evem.gov.si*, ki pa ni ista vrednosti *document.baseURI*, iz katere brskalnik sestavi absolutni URL. To vrednost pa smo imeli na voljo v Headless Chrome brskalniku, s katerim smo stran upodobili. Ker lahko s Selenium knjižnico na podoben način kot z BeautifulSoup potujemo po HTML dokumentu, zraven pa imamo na voljo že absolutni URL za vsako oznako, smo razčlenjevanje na koncu realizirali kar z uporabo Selenium-a in Headless Chrome.

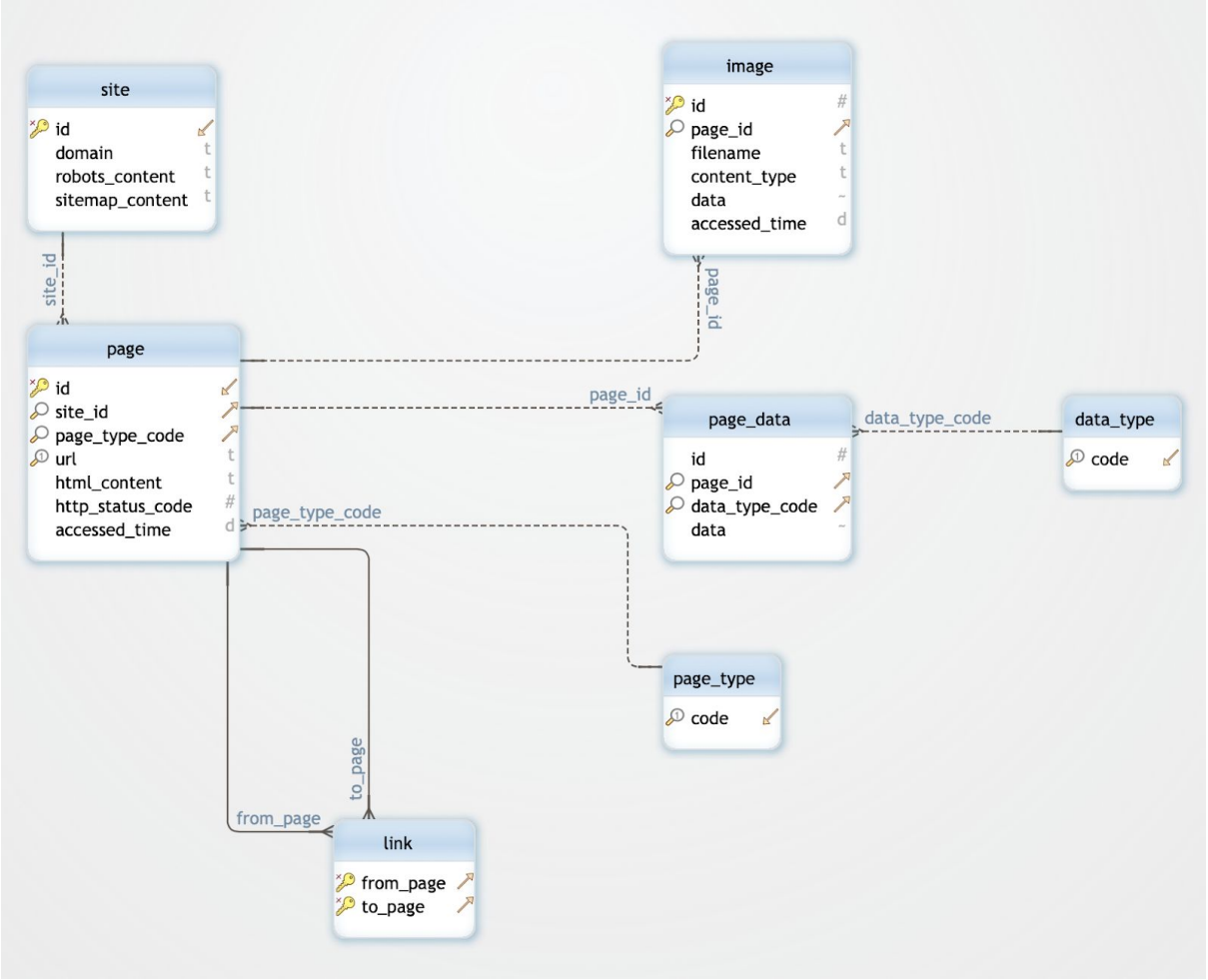
```
def parse_page(self, html_content):
    links = []
    images = []

    try:
        browser = self.driver

        anchor_tags = browser.find_elements_by_tag_name("a")

        for anchor_tag in anchor_tags:
            href = anchor_tag.get_attribute("href")

            url = self.get_parsed_url(href)

            if url:
                links.append(url)

        image_tags = browser.find_elements_by_tag_name("img")

        for image_tag in image_tags:
            src = image_tag.get_attribute("src")

            if src:
                image_url = self.get_parsed_image_url(src)

                if image_url:
                    images.append(image_url)
```

Za parsanje povezav iz Javascripta pa še vedno uporabljamo BeautifulSoup, saj je popolnoma ustrezen.
Vsako povezavo, ki jo pridobimo iz razčlenjene strani, moramo še vedno pregledati, saj lahko vsebuje razne akcije (povezave, ki se začnejo z tel: ali mailto:) ali javascript kodo (povezave, ki se začnejo z javascript:). Poleg tega moramo tudi odstraniti vse fragmente iz URL-ja (vse, kar se pojavi za znakom #), na koncu pa še zakodiramo posebne znake.

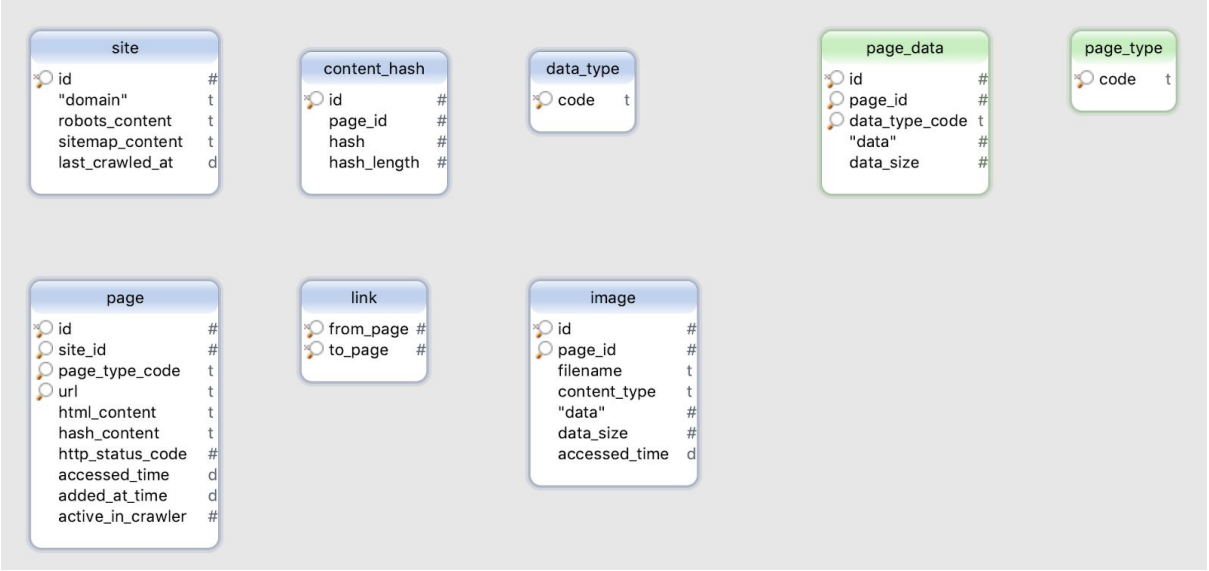
```
soup = BeautifulSoup(html_content, 'html.parser')

script_tags = soup.findAll('script')

for script_tag in script_tags:
    links_from_javascript = self.parse_links_from_javacript(script_tag.text)

    for link in links_from_javascript:
        links.append(self.get_parsed_url(link))
```

Pridobivanje povezav iz Javascripta

2.7. Oblikovanje baze



Začetna točka pri oblikovanju baze je bila podana predloga, prikazana na shemi zgoraj, v katero pa smo vnesli nekaj sprememb.
V tabelo *page* smo dodali stolpce *hash_content, added_at_time* in *active_in_crawler*. Z namenom realizacije FIFO vrste in posledično strategije breadth-first smo v atribut *added_at_time* vpisovali točen čas, ko je bila stran dodana v bazo oziroma frontier. Več o tem v rubriki *Kako smo realizirali strategijo breadth-first?*. Atribut *active_in_crawler* smo uporabljali za označevanje, ali se spletna stran trenutno obdeluje ali ne. Tako smo lahko preprečili, da bi dva procesa vzela v razčlenjevanje isto spletno stran. V stolpec *hash_content* pa smo vpisovali z algoritmom SHA-256 zgoščeno vsebino spletne strani. Ta atribut smo nato uporabili pri preverjanju duplikatov. Več o tem lahko preberete v poglavju *Zaznavanje dupliciranih strani*. Za zaznavo dupliciranih strani smo dodali tudi tabelo *content_hash*, ki jo sestavljajo atributi *page_id, hash* in *hash_length*, uporabljajo pa se pri metodi iskanja podobnih stani.
V tabelo *site* smo dodali atribut *last_crawled_at*, ki pove, kdaj je bila spletna stran nazadnje obiskana. Atribut uporabljamo za spoštovanje direktive *crawl-delay*, ki jo lahko podaja *robots.txt*. Več o tem, kako smo realizirali politiko, ki jo podaja *robots.txt*, opisujemo v poglavju *Branje in upoštevanje robots.txt*.
Tabelama *image* in *page_data* smo dodali atribut *data_size*, ki opisuje, kot pove tudi ime, velikost vsebine strani oziroma slike. Tako smo lahko nato omejili velikost posameznih tabel in v primeru, da je bil limit dosežen, preprečili dodajanje novih slik oziroma strani v bazo.
Spodnja slika prikazuje končni izgled naše sheme.



2.8. Branje in upoštevanje robots.txt

Zelo pomembna lastnost spletnih strani, ki jih mora vsak pajek upoštevati, je datoteka robots.txt. Python ima že vgrajeno knjižnico *RobotFileParser*, s katero lahko preberemo datoteko *robots.txt* in ugotovimo, ali lahko neko spletno stran pridobimo s pajkom ali ne. Sama knjižnica je že dokaj zastarela, saj ne podpira Sitemapov, prav tako pa lahko datoteko robots.txt pridobimo samo s klicem na strežnik. To pomeni, da nismo mogli uporabljati vsebine robots.txt, ki smo jo že pridobili za dotično domeno in tudi shranili v podatkovno bazo. Na srečo je knjižnica odprtokodna in dostopna na portalu github, zato smo jo lahko ročno uvozili v naš projekt in jo spremenili, da je zadovoljevala naše potrebe. Dodali smo

podporo za več Sitemapov, saj jih je lahko v robots.txt prisotnih več, in omogočili uvoz vsebine robots.txt iz naše podatkovne baze.
V samem pajku smo uporabili metode *read()*, ki prebere datoteko, *can_fetch(useragent, url)*, ki za podani url pove, ali ga lahko obiščemo in razčlenimo, in *crawl_delay(useragent)*, ki nam pove, koliko časa moramo počakati med zahtevki za isto domeno.

```
    """
        Checks if robots are set for the current site and if they allow the crawling of the current page
    """

    def allowed_to_crawl_current_page(self, url):
        if self.robots_parser is not None:
            return self.robots_parser.can_fetch('*', url)

        return True

    """
        Checks if crawl-delay property is set and if it exists check if the required time has elapsed
    """

    def wait_for_crawl_delay_to_elapse(self):
        try:
            if self.robots_parser is not None:
                crawl_delay = self.robots_parser.crawl_delay('*')

                if crawl_delay is not None:
                    if "last_crawled_at" in self.site and self.site["last_crawled_at"] is not None:
                        site_last_crawled_at = self.site["last_crawled_at"]

                        can_crawl_again_at = site_last_crawled_at + timedelta(seconds=crawl_delay)

                        current_time = datetime.now()

                        time_difference = (can_crawl_again_at - current_time).total_seconds()

                        if time_difference > 0:
                            print("     [CRAWLING] Crawl delay has not yet elapsed for site: {}".format(
                                self.site["domain"]))

                            time.sleep(crawl_delay)
        except Exception as error:
            print("     [CRAWLING] Error while handling crawl delay", error)
```

Implementacija podpore za robots.txt

2.9. Katere zunanje knjižnice uporablja spletni pajek?

- **Selenium WebDriver:drive**
  Selenium Webdriver sprejema ukaze in jih s pomočjo gonilnika izvede v brskalniku ter vrne rezultate. V našem projektu smo uporabljali *WebDriver* za brskalnik Chrome, s pomočjo katerega smo pridobili in razčlenili spletne strani.

- **Requests:**
  Requests je pythonska knjižnica za izvedbo HTTP poizvedb. Pri implementaciji našega pajka smo jo uporabljali za ukaz GET, s katerim smo pridobili vsebino iskane spletne strani.

- **Beautiful Soup:**
  Beautiful Soup je pythonska knjižnica, namenjena pridobivanju datotek HTML in

XML. V povezavi z izbranim razčlenjevalnikom (angl. parser) omogoča iskanje in navigiranje skozi razčlenjeno stran. Pri realizaciji našega spletnega pajka smo uporabili razčlenjevalnika *html.parser* in *lxml*.

- **RobotParser:**
  Odprtokodna knjižnica za Python 3, ki smo jo priredili za naše potrebe, kot je bilo opisano v poglavju 2.8 (original na voljo na povezavi https://github.com/python/cpython/blob/3.7/Lib/urllib/robotparser.py)

- **Multiprocessing:** (Process in Lock):
  Spletne strani pridobivamo z več procesi. V ta namen uporabljamo knjižnico *Multiprocessing* in njen objekt *Process*. Da ne bi prišlo do situacije, ko dva ločena procesa vzameta isto stran iz frontierja, uporabljamo tudi objekt *Lock*, ki zaklene funkcijo, da lahko do nje dostopa samo en proces. Več o implementaciji procesov lahko preberete v poglavju 2.2 Kako smo implementirali več workerjev.

- **Psycopg2:**
  Psycopg je vmesnik (angl. adapter) za uporabo PostgreSQL v Pythonu, oblikovan z mislijo na večnitne (angl. multi-threaded) aplikacije, ki sočasno (angl. concurrent) uporabljajo več kazalcev (angl. cursors) in izvajajo operacije, kot sta *INSERT* in *UPDATE*. Z uporabo razredov, ki nam jih ponuja vmesnik, recimo *pool*, *cursor* in *connection*, smo si olajšali iskanje, posodabljanje in vstavljanje zapisov v bazo.

- **Hashlib:**
  Za zgoščevanje, ki ga uporabljamo pri preverjanju duplikatov (več o tem lahko preberete v podpoglavju Zaznavanje dupliciranih strani) uporabljamo pythonski modul *Hashlib*. Hashlib omogoča uporabo različnih zgoščevalnih algoritmov, v našem primeru je to SHA-256. Funkcijo, ki iz dokumenta ustvari shingle, pa smo implementirali sami.

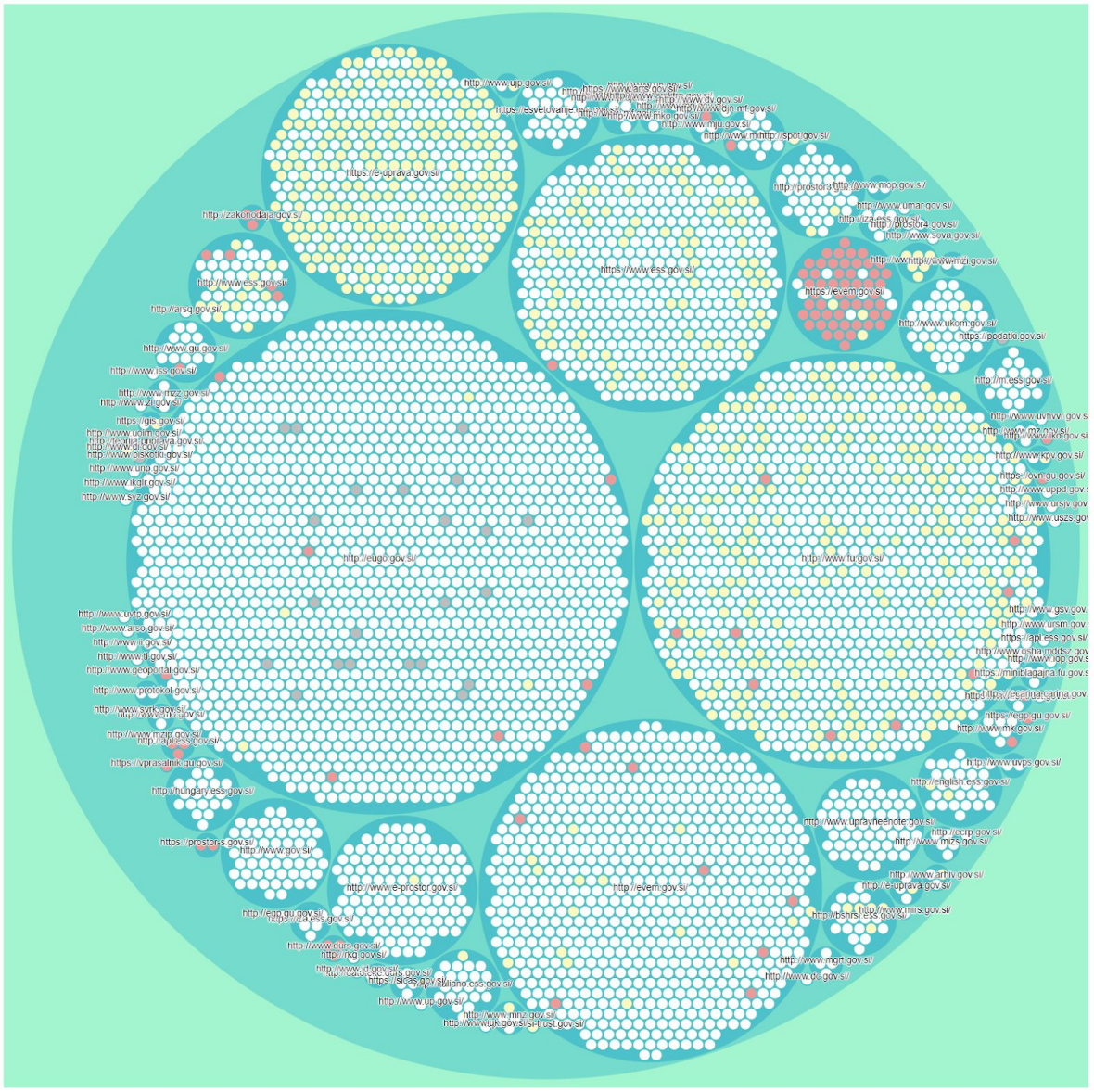
- **D3js:**
  D3js je knjižnica, napisana v JavaScriptu, ki je namenjena interaktivni vizualizaciji podatkov. Omogoča različne načine prikazov, kot so mehurčki (bubble chart in bubble map), dendrogrami, drevesa, besedni oblaki, grafi, zemljevidi itn. Več o tem, kako smo mi v naši projektni nalogi vizualizirali pridobljene podatke, si lahko preberete v rubriki Statistika in vizualizacija.

2.10. Statistika in vizualizacija

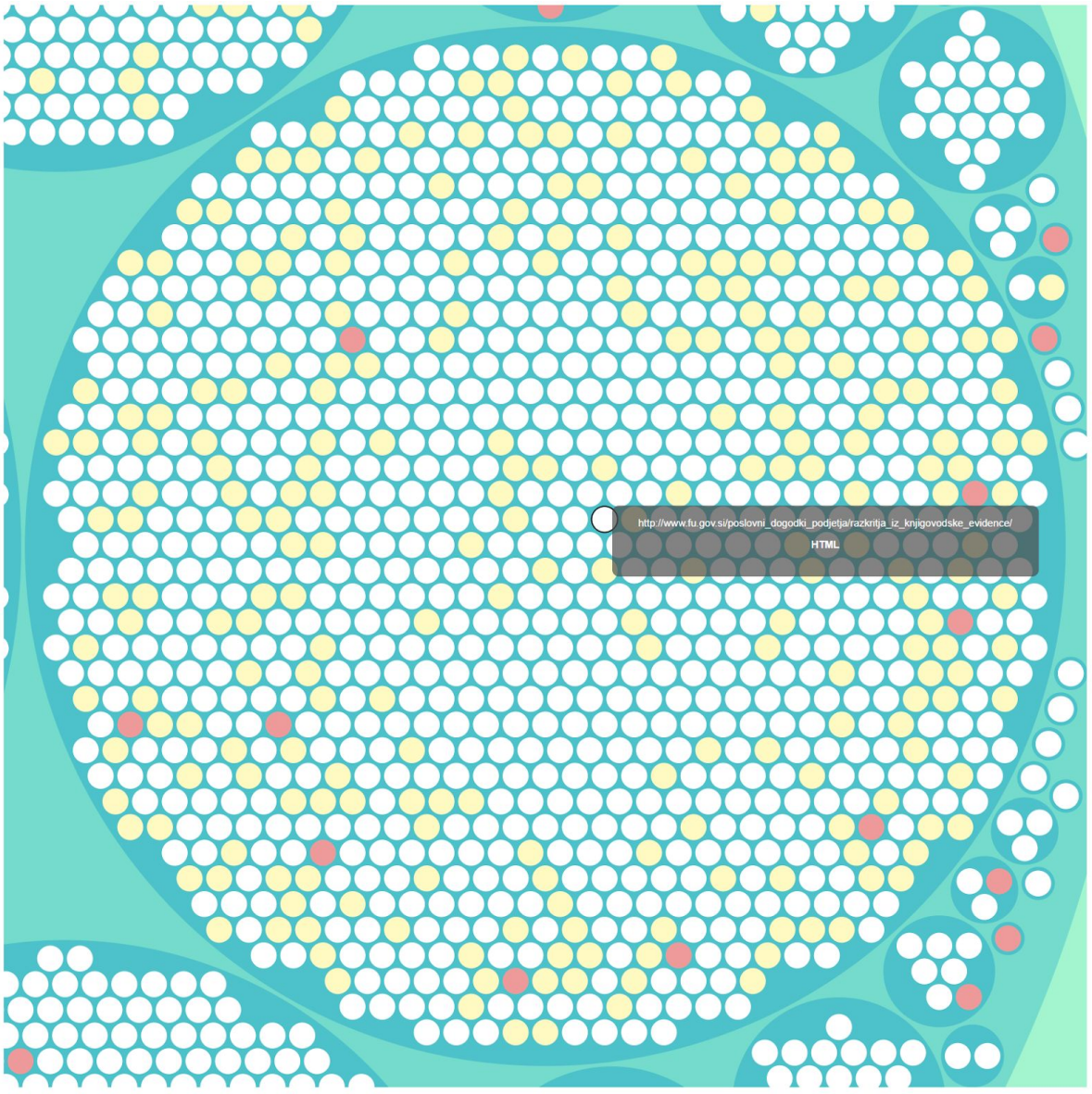
Podatke smo vizualizirali s pomočjo Javascript knjižnice D3.js. Ta v osnovi deluje tako, da vsebini strani (DOM-u) pripne element <svg>, kateremu so nato podrejeni različni navidezni in grafični elementi, nad katerimi nato s pomočjo programskega jezika Javascript izvajamo različne ukaze, ki spreminjajo prikaz (npr. drag and drop, zoom, itd.). Za naše potrebe smo izdelali dve vizualizaciji - hierarhični prikaz strani po domenah in povezave med posameznimi stranmi znotraj domene.

Na začetku smo morali podatke za prikaz preoblikovati v ustrezen format (json), ki je razumljiv knjižnici D3.js in na podlagi katerega ta pravilno prikaže vizualizacijo na zaslonu. Za potrebe predprocesiranja podatkov smo napisali enostavno python knjižnico, ki glede na podane parametre iz podatkovne baze izlušči ustrezne podatke, jih zapiše v json datoteko in le-to shrani v ustrezno mapo, kjer se nahajajo ostale datoteke za vizualizacijo. Z namenom tekočega delovanja vizualizacije in jasnega prikaza podatkov v grafih, smo število elementov v posamezni vizualizaciji ustrezno omejili.

Prvi graf, ki smo ga izdelali, prikazuje povezave med korenskimi stranmi in njihovimi podstranmi. Za prikaz smo uporabili t.i. pack način prikaza v obliki krogov na večih nivojih. Na prvem nivoju so prikazane domene iz katerih je pajek pobiral posamezne podstrani. Te so prikazane znotraj večjih krogov, kot manjši krogi. Iz grafa lahko ločimo različne tipe strani tako po vsebini kot po statusu zapisa v podatkovni bazi. Z rdečo barvo so pobarvane tiste strani, pri katerih je pajek naletel na težave (označene kot 'ERROR'). Z rumeno barvo so pobarvane tiste strani, katere je pajek prepoznal kot podvojene ('DUPLICATE'). S sivo barvo smo označili strani, ki so bile v robots.txt zapisane kot prepovedane za pregledovanje. Vse ostale strani so pobarvane z belo barvo. Za potrebe tekočega delovanja in jasnega prikaza, smo število strani omejili na malo več kot 3500. Z levim klikom lahko približamo pogled na ožjo skupino strani, z desnim klikom na posamezno stran (krogec), se povezava do strani odpre v novem oknu.

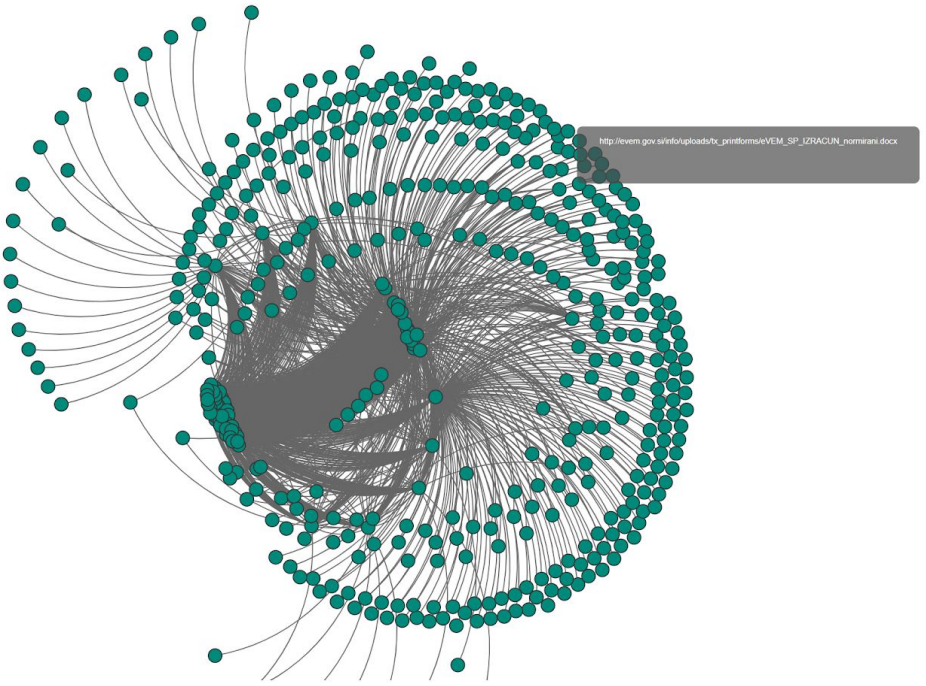


Prikaz korenskih strani in njihovih podstrani



Podrobnejši pogled na eno izmed korenskih strani, ki vsebuje več podstrani. Strani z različnimi statusi so obarvane z različnimi barvami, ob prekrivanju miške s posameznim krogcem/stranjo, se prikažejo informacije o dotični strani.

Nato smo izdelali še vizualizacijo, ki prikazuje povezovanje posameznih strani med seboj (kazanje ene strani na druge). Za prikaz smo uporabili network način prikaza, ki se uporablja za prikaz relacij med entitetami. Vizualizacija prikazuje povezave med posameznimi podstranmi domene ‘http://evem.gov.si/’ .Krogec v grafu predstavlja eno stran, med posameznimi stranmi pa so izrisane povezave, kakor so strani povezane med seboj. Za potrebe tekočega delovanja in jasnega prikaza, smo število strani omejili na 1500. Z desnim klikom na posamezno stran (krogec), se povezava do strani odpre v novem oknu. Graf po zaslonu lahko premikamo z vlečenjem posameznih točk.



Prikaz povezave med stranmi domene http://evem.gov.si/. Ob prekrivanju miške s posameznim krogcem/stranjo, se prikažejo informacije o dotični strani.

Najdaljše časovno obdobje poganjanja spletnega pajka je v našem primeru trajalo približno dva dni. V tem času je programu s tremi simultanimi procesi uspelo pregledati približno 92.000 strani. Podatke smo izvozili v obliki sql datoteke, ki se nahaja na povezavi: https://drive.google.com/file/d/13DhB0wn-z1iV-SdVczx6DfLpZzdExWrS/view?fbclid=IwAR0UZnKfXSDg_8yYtEhkWK5T-bz8aJlNEe3kYt79YHkedpFneMazdVffcy0. V datoteki se nahajajo samo podatki o vseh straneh, medtem ko podatki o povezavah in korenskih straneh niso vključeni, saj smo pri izvažanju očitno storili napako, kar smo opazili šele potem, ko smo izvorne podatke iz podatkovne baze že izbrisali.

3. Viri

- Shikhar Gupta, 29. junij 2018: Locality Sensitive Hashing; An effective way of reducing the dimensionality of your data, https://towardsdatascience.com/understanding-locality-sensitive-hashing-49f6d1f6134

4. Zaključek

S končnim izdelkom smo zelo zadovoljni, saj smo z izdelanim spletnim pajkom dosegli naša pričakovanja, hkrati pa smo se ob razvoju naučili veliko novega.