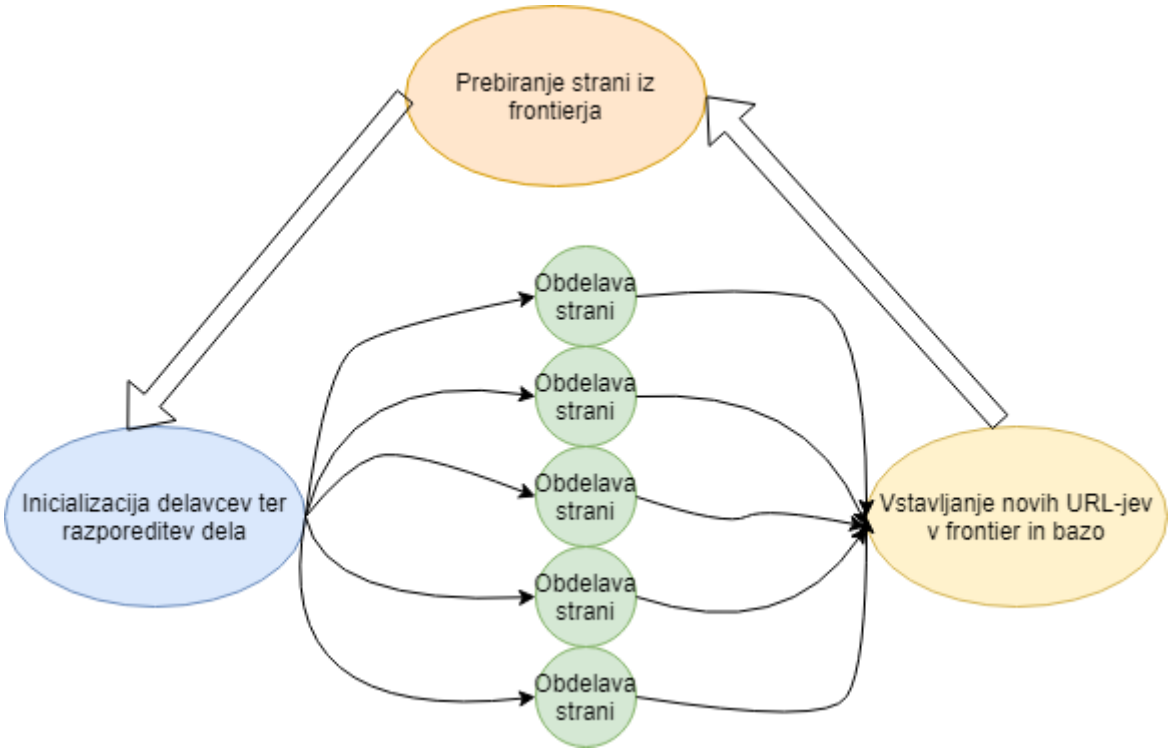
DELOVANJE PAJKA

Naša naloga je bila da napišemo spletnega pajka, ki naj se sprehodi po spletnih straneh domen ki se končajo z *.gov.si*. Pajek mora delovati na več nitih hkrati, znati interpretirati tako *html* kot *javascript* kodo, najti ter shraniti datoteke določenih tipov (vključno z slikami), detektirati duplikate (torej spletne strani ki jih je pajek že obiskal) ter voditi evidenco o vsem skupaj v podatkovni bazi. Pri obiskovanju spletnih strani mora pajek upoštevati datoteko *robots.txt*.

1.) Splošen pregled

Implementacijo pajka smo razdelili na 4 dele. Prebiranje strani iz frontierja, inicializacija delavcev ter razporeditev dela, obdelava spletnih strani ter vstavljanje novih URL-jev ter datotek v bazo. Prva dva dela bomo podrobneje predstavili v tem poglavju, druga dva pa v sledečih poglavjih. Za lažje razumevanje delovanja pa bomo na začetku konceptualno predstavili celotno delovanje pajka.



Na začetku se frontier napolni s stranmi, podanimi ob zagonu programa. Glede na zahtevano število delavcev se izvede inicializacija, kjer vsak delavec prejme 1 spletno stran. Nato vsak delavec vzporedno z ostalimi obdela svojo prejeto spletno stran. Pajek počaka, da delavci zaključijo delo, nato pa rezultate glede na njihovo vsebino ustrezno zapiše nazaj v frontier ali v bazo. Cikel se ponavlja, dokler frontier ni prazen.

2.) Prebiranje strani iz frontierja ter inicializacija delavcev

Celoten pajek je implementiran s pomočjo zanke while, ki se, kot je omenjeno zgoraj, izvaja, dokler se frontier ne izprazne. V kakšnem vrstnem redu obiskujemo spletne strani v frontierju definira, na kakšen način bomo raziskovali splet. V naši implementaciji pajka je frontier realiziran s pomočjo pythonovega seznama (eng. List). Nove strani dodajamo na konec seznama, strani za obiskovanje v trenutni iteraciji pa pobiramo z začetka seznama. Frontier torej deluje kot povsem običajna FIFO vrsta. Taka implementacija frontierja narekuje Breadth First Search, torej iskanje v širino, kjer najprej obiščemo vse spletne strani na prvem nivoju, šele potem začnemo raziskovati strani na drugem nivoju.

Ko pajek prebere spletne strani iz frontierja mora najprej preveriti, če za domeno vsake od strani obstaja robots.txt. Ta nam narekuje, kako pogosto lahko to domeno obiskujemo ter katere povezave s strani lahko obiščemo, katere pa ne. Vsebuje tudi sitemap, ki smo ga prav tako morali shraniti v bazo. Do robots.txt smo dostopali s pomočjo python knjižnice requests, dobljeno vsebino pa smo obdelali s pomočjo knjižnice url.lib.

Opremljen s podatki iz robots.txt lahko pajek inicializira več delavcev. Točno število delavcev izbere uporabnik ob zagonu programa. Delavci so implementirani s pomočjo niti iz python knjižnice threading. Naloga vsake posamezne niti je, da obišče prejet URL naslov, s spleta sname celotno HTML vsebino naslova URL in jo pregleda za nove povezave, slike ter binarne datoteke. Nato mora pridobljene nove naslove URL obdelati, da so primerni za vstavljanje v frontier. Za obiskovanje spletnih strani smo uporabili python knjižnico selenium, ki omogoča izvajanje javascripta, brez da bi zagnali brskalnik in smo tako lahko obdelali tudi tiste strani, katerih vsebina je skrita v različnih skriptah. Nadaljnja obdelava strani je opisana v naslednjem poglavju.

Vse niti svoje rezultate vrnejo v vrsto, ki je del knjižnice multithreading. Pajek torej počaka, da vsaka nit zaključi z delom nato pa iz vrste vzema rezultate ter jih vstavi v frontier in v bazo. S tem se zaključi 1 cikel delovanja.

3.) Obdelava strani

Razred, ki implementira obdelavo spletnih strani, se imenuje “htmlGetAll” in vsebuje tri metode, ki so ključne za delovanje - “urlCleaner”, “possiblyExtendUrl” in “doPage”. Metode poskrbijo, da se iz spletne strani izluščijo vsi URL naslovi (in popravijo, če je to potrebno), slike in ostale binarne datoteke.

Iz modula bs4 je bilo potrebno vključiti BeautifulSoup, ki nudi funkcionalnosti pri iskanju značk po html dokumentu. Potrebovali smo še posixpath za normalizacijo URL poti, modul re za regularne izraze in več metod (urlparse, urlunparse, urljoin, quote, unquote) iz modula urllib.parse za pomoč pri obravnavi samih URLjev.

Vsako stran obdelamo z metodo “doPage” ki prejme tri parametre - “base_url” (osnovni url), “htmlSoup” (html trenutne strani) in “robots” datoteko. Znotraj te metode pridobimo vse URLje (bodisi tiste v html-ju bodisi tiste v javascript funkcijah), vse slike (tiste, ki se nahajajo pod značko “img”) in datoteke (gledamo končnice: .pdf, .doc, .docx, .ppt, .pptx).
Prvo pogledamo, če ima mogoče html definiran osnovni URL, če ga ima, uporabimo tega, drugače pa ostane isti, kot je bil podan. Nato se sprehodimo s for zanko čez vse “a” značke in pridobimo vrednost iz “href”. To vrednost nato podamo metodi “possiblyExtendUrl”, ki nam vrne absolutni kanoniziran URL. Ta URL nato dodamo v množico URL povezav (le če vsebuje “gov.si") ali v množico datotečnih (binarne datoteke) povezav - zavisi od končnice URLja.

Ko obdelamo vse “a” značke, se sprehodimo še čez lokacije v javascript funkcijah (najdemo jih s pomočjo naslednjega regex izraza: "document.location.href *= *[‘\”].*[‘\”]"). Povezave obdelamo na podoben način kot v prejšnji for zanki.

Sledi še for zanka čez vse “img” značke v htmlju iz katerih pridobimo povezave do slik.

Metoda na koncu vrne množico URL povezav (brez osnovnega URLja strani na kateri smo pridobili vse te podatke), množico datotečnih (binarne datoteke) povezav in množico povezav do slik.

Tudi vse URLje je bilo potrebno posebej predelati. Najprej vsak URL obdelamo z metodo urlCleaner, ki na vhod prejme URL in vrne sprocesiran oziroma kanoniziran URL.

Najprej URL podamo metodi urlparse, ki razstavi URL na več imenovanih delov - “scheme”, “netloc”, “path” itn. Iz tega objekta nato vzamemo “path” in jo s pomočjo funkcije posixpath.normpath normaliziramo. Če je pot “/“, ta funkcija vrne “.” in zato pot v tem primeru zamenjamo z “/“. Sledi odstranitev “default file-a”, kot je index.html ali index.php. To storimo s preprosto zamenjavo tega dela niza s praznim nizom (re.sub(defaultFileName, ‘’).
Nato preverjamo ali se pot konča z naslednjimi končnicami “\.\w*$”. Slednji izraz zajame vse besedne končnice. Če se pot ne konča s končnico in če na koncu nima “/“, potem ji dodamo “/“, saj gre očitno za direktorij (kot navajajo pravila s 16. “slajda” na prosojnicah s predavanj). S pomočjo unquote in quote nato prvo dekodiramo vse zakodirane znake v URL in jih spet zakodiramo, a tokrat se zakodirajo le tisti znaki, ki so potrebni (metodi quote podamo pod argument “safe” naslednje znake “:_.-/~”, ki naj se ne zakodirajo). Sledi procesiranje “netloc” atributa. Če je prisoten, potem imamo opravka s celotnim URLjem in moramo iz njega odstraniti številko vrat in spremeniti črke v male črke. Če ni priosten, nam tega ni potrebno storiti. Na koncu metoda urlCleaner le še zapakira (s pomočjo urlunparse) sprocesiran URL brez fragmentov in ga vrne kot rezultat.

Nato URL obdelamo še z possiblyExtendUrl. Ta metoda poskrbi za potrebno razširitev relativnega URL-ja v absolutnega, če je to potrebno. Na vhod prejme “baseUrl” (osnovni oziroma bazni URL), “relUrl” (relativni URL) in “robots” (URLji do katerih ne smemo dostopati). Prvo sprocesiramo relativni URL in če je “scheme” enak nizu javascript ali je celotni relativni URL prazen niz ali prisoten v “robots”, potem vrnemo kar osnovni URL. Nato preverimo, če je v relativnem URLju prisoten “netloc”. Če ni, potem združimo osnovni URL s sprocesiranim relativnim URLjem (kanoniziran z urlCLeaner funkcijo) s funkcijo urljoin in vrnemo rezultat. Če je “netloc” prisoten, imamo opravka z absolutnim URLjem in vrnemo le sprocesiran URL (kanoniziran).

4.) Zapisovanje v bazo

Podatkovno bazo PostgreSQL smo postavili na strežniku do katerega smo imeli dostop vsi. Na ta način delovanje pajka ni bilo omejeno na en sam računalnik, kar je bilo pomembno saj imamo v skupini le vsak svoj prenosnik. Z nekaj modifikacijami bi lahko pajek tekel tudi na več računalnikih hkrati.

Edina modifikacija predloženi shemi je stolpec *content_hash* v tabeli *page.*

Za dostop do baze z Pythonom smo si pomagali z knjižnico psycopg2 ter spisali lastni modul *database.py* s pomočjo katerega smo opravljali vso interakcijo z bazo.

Ko sprva zaženemo pajka najprej v bazi preverimo če tabela *page* že vsebuje kako stran z oznako '*FRONTIER*', ter jih prenesemo v RAM, torej v lokalno kopijo frontirja. Prav tako si v lokalni spomin prenesemo URL-je ter vsebino vseh že obiskanih strani, v namen detekcije duplikatov. Več o detekciji duplikatov kasneje. V primeru da je v bazi frontir prazen dodamo vanj začetne strani (http://evem.gov.si, http://e-uprava.gov.si, http://podatki.gov.si, http://e-prostor.gov.si).

Ko pajek obdela neko stran, podatke o njej zapiše v bazo v treh fazah. Najprej posodobi ustrezno vrstico v tabeli *page*: doda vsebino html datoteke ter njeno zgoščeno vrednost, http status, čas dostopa ter posodobimo tip strani (iz *'FRONTIER'* v *'HTML'* oziroma *'DUPLICATE'*). Nato doda vse povezave ki jih najde na strani v frontir, tako lokalno kot v bazi. Pri tem mora biti pazljiv da doda tudi morebitne nove domene v tabelo *site*. Vsako povezavo doda tudi v tabelo *links*. Na koncu doda v bazo še vse slike ter binarne datoteke.

Detektiranje duplikatov poteka v dveh korakih. Če naletimo na URL ki je identičen že obiskanemu potem ga sploh ne dodajamo v frontir. Zgodi pa se da pod drugim URL pridemo na stran ki je identična po vsebini. Primerjanje po celtoni html datoteki bi bilo zamudno, zato raje primerjamo po

zgoščeni vrednosti html kode. Zgoščevanje je impelmentirano z algoritmom md5 in traja v povprečju približno eno sekundo, v zameno pa prihranimo čas pri vsaki primerjavi.

STATISTIKA

Prva tabela prikazuje najbolj osnovne podatke o rezultatih pajka. V celotni dobi delovanja je pajek obiskal 216 glavnih domen ter obdelal 4375 strani, od katerih je bilo 217 duplikatov. V frontier je dodal še 19590 strani, ki jih je bilo potrebno obiskati, vendar smo zaradi oddaje iskanje zaključili. V povprečju je vsaka glavna domena vodila v 20 podstrani.

	Number of sites	Number of pages	Number of duplicates	Number of pages still in frontier
Baza	216	4375	217	19590
Začetne 4 strani	4	210	4	0

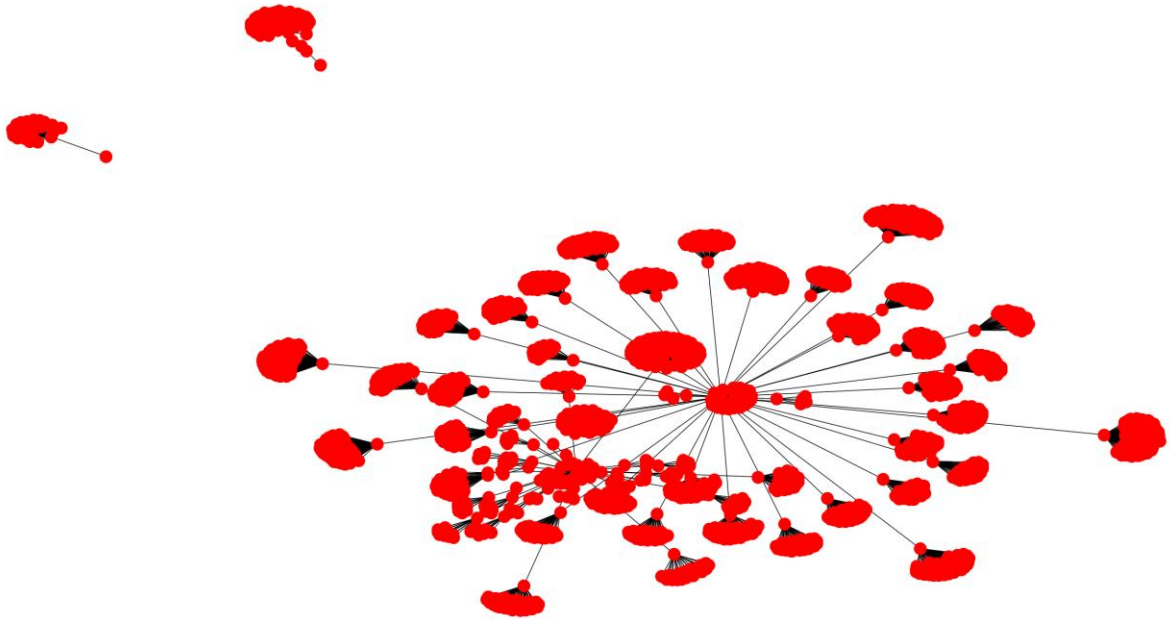
Druga tabela prikazuje nekaj podatkov o slikah, ki jih je pridobil pajek. Najpogosteje je pajek naletel na slike tipa jpg samo enkrat pa je naletel na sliko tipa svg. Vidimo, da je enkrat prenesl tudi HTML datoteko. V povprečju je vsaka obiskana stran vsebovala manj kot 1 sliko.

	PNG	GIF	SVG	JPG	JPEG	HTML
Baza	319	270	1	888	10	1
Začetne 4	0	0	0	0	0	0

Tretja tabela prikazuje število različnih binarnih datotek, ki jih je pajek našel med preiskovanjem spleta. Vidimo lahko, da je Pajek največkrat naletel na PDF-je, najmanjkrat pa na powerpoint datoteke. V povprečju je pajek na vsaki strani našel manj kot 1 binarno datoteko.

	PPTX	DOCX	PDF	PPT	DOC
Baza	9	175	1426	9	335
Začetne 4	0	0	0	0	0

Na koncu podajamo še vizualizacijo obiskanih strani. Vizualizacijo smo izvedli s pomočjo python knjižnic networkx in matplotlib. Slika prikazuje pomanjšanj graf, saj je bilo celotno omrežje preveliko, da bi ga vključili v sliko.
Opazimo, da iz vsake strani izhaja več drugih strani, vendar pa je zelo malo prepletanja med najdenimi stranmi. Temu je tako, ker je veliko strani ostalo v frontierju in so še neraziskane.



Zaključek

Pajka bi lahko še precej izboljšali. Najprej bi lahko preverili če posamezna stran sploh vsebuje *javascript* elemente, in v primeru da jih ne Seleniuma sploh ne bi uporabili, ter celotni html dokument precej hitreje prenesli z knjižnico requests.

Celotna zasnova večnitnega delovanja ni optimalna, saj vse niti hkrati koristijo internetno linijo ter nato hkrati koristijo CPU, medtem ko je glavna ideja uporabe večih niti ravno ta, da se različni resourci koristijo ob različnih časih.

Še najbolj učinkovito bi bilo, če bi sploh ne čakali da se posamezne niti združijo po vsaki iteraciji. Vsaka nit bi torej iz skupnega frontirja jemala spletne strani, jih obdelala, vstavila podatke v bazo ter nato ne čakala da se terminirajo še ostale niti temceč takoj nadaljevala z naslednjo stranjo iz frontirja. Tako bi se znebili tako čakanja na najpočasnejšo nit kot tudi precej overheada kot je zaganjanje niti, zaganjanje vedno novih instanc Seleniuma itd.

Končna izboljšava bi bila dopolnitev dostopanja do baze z ustreznimi ključavnicami, tako da bi lahko pajek tekel na več računalnikih hkrati. Ena naših težav je bila da je pajek tekel samo ponoči, ko nismo uporabljali svojih prenosnikov. Precej bi pomagalo če bi lahko takrat koristili vse tri računalnike hkrati.

Precej overheada se najbrž nabere tudi z manj-kot-idealno zasnovo programa, ki je neizbežen stranski produkt kolaboracije brez neke centralne vlogo katere naloga je inkorporacija modulov v glavno ogrodje programa. V kontrast, mi smo več ali manj samo nalagali novo kodo prek že obstoječe.