Математички факултет у Београду

Мастер рад

Примена програмског језика Python у реализацији алгоритама за рангирање веб страница

Аутор:
Игор Илић

Ментор:
Проф. др Владимир Филиповић

Београд, 2015

Садржај

1 Увод

Током развоја Интернета и поготово од настанка веба, повећавала се потреба за претраживањем страница које су биле постављене на Интернет. Са настанком веба и његовим ширењем та потреба се продубила због огромног броја информација које су преплавиле веб од настанка до данас. Највећи проблем са којим су се корисници Интернета, да се из мора информација издвоји она која је најподеснија за корисников упит. Овај рад покушаће да реши проблем претраживања веб страница, кроз реализацију алгоритма за рангирање веб страница у програмском језику Python. Такође, у току тог процеса, биће анализиран алгоритам за рангирање PageRank™ [1] и његове предности у односу на претраживање без примене датог алгоритма.

Рад ће бити подељен на неколико делова. У првом делу биће укратко објашњена синтакса и рад програмског језика Python. Обратиће се посебна пажња на основне елементе Python-а, као и на начин писања програма у њему. Такође, уз навођење једноставних примера, биће обрађени и типови података који се користе у Python-у, начин на који се уводе услови, петље, функције и модули, као и на који начин Python приступа класама и објектима.

После дела о програмском језику Python биће приложен део о претраживању Интернета и обради хипервеза. Биће приложен и пример како се из HTML кôда издвајају хипервезе, што ће чинити основу веб-паука.

Затим се описује реализација веб-паука, који скупља све хипервезе на веб страници и потом следи те хипервезе да би наставио свој процес унедоглед или до дубине претраживања који му се одреди.

После тога долази се до процесирања корисниковог упита и описује се како ће претраживач одговарати на упите. Поставља се питање да ли је процесирање могуће радити на бржи начин и предлаже се коришћење хеш табеле.

На крају, рад се бави питањем рангирања веб страница у процесу претраживања. Анализира се алгоритам PageRank™ даје се могуће решење овог алгоритма у програмском језику Python.

После закључка, дати су и додаци који садрже комплетан кôд, затим поређење резултата приликом задавања модела претраживања са и без укљученог модула за рангирање страница. Такође, дат је списак програмских кôдова који су се појавили током писања овог рада, као и литература која је коришћена током писања рада.

У примерима који се налазе у мастер раду је коришћена верзија Python-а 2.7.6, мада би примери требали да раде и са новим верзијама. Примери су тестирани у програмском окружењу које чини едитор *Vim*, уз одговарајуће додатке, а у оквиру оперативног система Ubuntu Linux 14.04.

[1] PageRank™ алгоритам није јавно доступан, те ће се његова анализа свести на оно што јесте јавно доступно и на анализе других аутора, нпр. [**?**]

2 Програмски језик Python

2.1 Карактеристике и историјат

2.1.1 Програмски језици и програмирање

Програмирање подразумева активност *програмера* усмерена ка решавању конкретног проблема на рачунару. Проблем се обично решава конструкцијом *алгоритма* за решење проблема, а алгоритам се записује посредством програмског језика. Програмски језик има улогу да обезбеди конструкције и начине за израчунавање на рачунару. Организовано израчунавање обично називамо *програмом*.

Програмски језик је средство за писање програма и саопштавање програма рачунару. То је вештачки језик који првенствено служи за комуникацију између човека и рачунара, мада се понекад програмски језик користи и за комуникацију између људи.

До сада је направљено неколико хиљада програмских језика и потребно је да се на неки начин разврстају. Постоје разне класификације, зависно од *критеријума класификације*:

- степен зависности од рачунара
- време настанка и својства
- област примене
- начин решавања проблема

Тако по степену зависности од рачунара програмске језике можемо поделити на машински зависне (Асемблерски језици, Макро-језици, итд.) и машински независне програмске језике.

По областима примене програмске језике можемо поделити на оне за учење програмирања (Pascal, Basic, Logo, итд.), програмске језике за развој системског софтвера (C, Asembler, итд.), за пословну примену (COBOL, SQL, ...), за развој програма на интернету (Java, Java Script, Perl, итд.), за примену у математици, и друго.

Програмске језике делимо и по начину решавања проблема и то на процедуралне и непроцедуралне. Док би их по времену настанка и својствима поделили на језике I генерације (машински и асемблерски), II генерације (FORTRAN, COBOL, LISP, BASIC,...), III генерације (Pascal, C, PROLOG, Smalltalk, C++, Java, ...) и језике IV генерације (SQL, VisualBasic, сви језици унутар апликација).[**?**]

Може се поставити и питање *зашто* учити компјутерско програмирање? Програмирање развија креативност, логичко размишљање и способност решавања проблема. Програмер добија прилику да креира *нешто* из *ничега*, користи логику да претвори скупове речи у компјутерски програм и кад нешто не иде како треба на дело долази способност да се реши проблем и да се нађе и исправи грешка.

Програмирање је и забава. Понекад је то и захтевна (и с времена на време фрустрирајућа) активност, али где се те способности могу искористити како у школи и на послу, тако и у кући невезано за захтеве каријере и образовања.

2.1.2 Зашто Python?

Чињеница је да постоје на хиљаде програмских језика. Неки од њих су поменути у претходном делу. Шта то издваја програмски језик Python од других и зашто би се неко одлучио да програмира у Python-у, уместо, на пример, у Pascal-у?

Python је лак за учење, а уз то има заиста корисна својства за програмера почетника. Кôд је врло разумљив за читање, ако се упореди са осталим програмским језицима. Такође, постоје разни додаци (модули) који ће проширити Python на захтевани начин, тако да Python може да послужи за креирање једноставних анимација са додатком Turtle (инспирисан Turtle graphics, коришћеним у програмском језику Logo у шездесетим годинама прошлог века)[?]. Други модули служе да користимо Python као скрипт језик или као додатак разним комплекснијим језицима попут C/C++.

Надаље, Python може представљати сјајно оруђе за учење основних појмова програмирања, типова података, услова, петљи, за учење разних алгоритама(на пример сортирања), онда за имплементацију математичких формула и начина на који се те формуле могу програмирати (итеративно или рекурзивно, на пример за факторијел).

Важан фактор је такође и *доступност* Python-а, јер се интерпретатор може врло једноставно преузети са интернета и инсталирати на рачунару. Постоје и разне варијанте Python интерпретатора које се налазе на појединим веб страницама. Python може да ради на многобројним оперативним системима: Windows-у, Mac OS-у, Linux-у, па и на мобилним платформама iOS-у или на Android оперативном систему[?] .

2.1.3 Историјат и начин коришћења

Програмски језик Python је настао почетком '90-их година прошлог века. Креирао га је Холанђанин Гвидо ван Росум (Guido van Rossum) и наставио да га развија до данашњег дана, наравно уз помоћ огромне заједнице програмера широм света. Данас је актуелна верзија 3.4. Гвидо је у свету познат под надимком BDFL, што је скраћеница за *Доживотни Добронамерни Диктатор*[2].

Python је добио име по култној британској хумористичкој серији у продукцији BBC-а, "Летећи циркус Монти Пајтона" (*Monthy Python's Flying Circus*), коју су прославили легендарни глумци и комичари попут: Ерика Ајдла, Џона Клиза, Мајкла Пелина, Грејема Чепмена, режисера Терија Гилијама и других. Снимљена као алтернативни театар апсурда, серија је остала позната и у данашње време и до данас се памте и радо гледају скечеви попут "*Министарства смешног ходања*", "*Најсмешнији виц на свету*", итд.

Оно што је још остало је да се опише на који начин може да се користи Python на рачунару. Дакле, потребан је Python интерпретатор, који се може бесплатно преузети са званичне Python-ове веб презентације[?], као и неки од текст едитора и/или развојних окружења (нпр. *Eclipse*, *PyCharm*, *Vim* или било који текст едитор) помоћу кога ће се уносити кôд. Алтернативно може се преузети и инсталирати апликација *IDLE* (`http://www.python.org/getit/`), која омогућује да се програмира у Python-у на врло једноставан начин.

[2]Benevolent Dictator For Life

2.2 Рачунање и променљиве

У даљем тексту се претпоставља да је Python инсталиран на рачунару и апстрахује се начин на који се добија резултат на потенцијалном екрану или принтеру. Дакле, описује се искључиво програмски језик Python, његова синтакса, као и шта се све може да урадити помоћу њега.

2.2.1 Python као калкулатор

Python може да послужи и као обичан калкулатор[**?**, **?**]. Ако је потребно да се изврше основне аритметичке операције: сабирање, одузимање, множење и дељење, треба користити адекватне симболе за дате операције (редом): +, -, *, /. Поставља се питање како се Python односи према разним типовима бројева. Бројеви у рачунарству се обично деле на целобројне и бројеве у покретном зарезу. За операције сабирања, одузимања и множења правило је врло једноставно: ако се ради са целим бројевима, добија се и целобројни резултат. Исто важи и за бројеве у покретном зарезу. Мало је другачији случај са дељењем. Ако се врши дељење целих бројева, количник ће бити дат као децимални број. Ако треба да се као резултат добије цео део количника, користи се симбол "//". На пример:

```
>>> 100 + 16
116
>>> 96.0 - 11
85.0
>>> 11.1 * 10
111.0
>>> 20/5
4.0
>>> 20 / 8
2.5
>>> 20 // 8
2
```

Кôд 1: Примери операција са бројевима

Могу се користити и друге математичке операције, као што су степеновање или се може направити програм за произвољну математичку функцију. Степеновање је другачије него што је то у осталим програмским језицима и обавља се помоћу симбола "**":

```
>>> 2 ** 32
4294967296
```

Кôд 2: Степеновање

Такође, Python омогућава и рачунање са комплексним бројевима у алгебарском облику, где се код комплексних бројева користи суфикс "j" или "J" ако треба да се назначи имагинарни део тог комплексног броја. На пример:

```
>>> (1+1j)*(2-1j)
(3+1j)

>>> 1j ** 2
```

```
(-1+0j)
>>> 1j * 1J
(-1+0j)
>>>
```

Кôд 3: Операције са комплексним бројевима

2.2.2 Промењиве

Променљивој се додељује одговарајућа вредност, која може бити бројчана или логичка (тачно или нетачно) или променљива може да реферише на вредност друге променљиве. Ако је ово последње случај, промена вредности једне променљиве, утиче и на вредност друге променљиве. У примеру који следи, наредбом:

```
>>>a = 1
```

Слика 1: 1

променљива *a* има вредност 1. Може се рећи и да променљива реферише на вредност 1 која се налази негде у меморији.

Ако се истој променљивој додели нека друга вредност, на пример 2, онда ће она реферисати на број 2, док је број 1 напуштен.

```
>>>a = 2
```

Слика 2: 2

Потом може да се уведе нова променљива и да се подеси да реферише на променљиву *a*.

```
>>>b = a
```

Тада и једна и друга променљива реферишу на исту вредност. Ако би се једној од њих променила вредност, тада би друга реферисала на нову вредност.

Имена промењиве се по договору пишу свим малим или свим великим словима латинице, иако Python дозвољава све начине писања промењивих, осим што нису дозвољене кључне

Слика 3: и_реферишу на исту вредност

речи. Прави Пајтонисти[?] кажу да им је само име "промењљива" неприкладно за оно што она ради у Python-у, те да би бољи избор био употребити речи: "имена", "објекти" или "везивања" (*енгл., bindings*).

Промењљиве се могу користити и у рачунању. На пример:

```
>>> br_stanovnika = 9860000 # br. stanovnika Srbije prema proceni iz 2010.
>>> povrsina = 77474 # povrsina Srbije u kv. kilometrima
>>> gustina_naseljenosti = br_stanovnika / povrsina
>>> print (gustina_naseljenosti)
127.26850298164545
```

Кôд 4: Пример коришћења промењљивих

Дакле, могло би се рећи да програмски језик Python садржи функционалност моћног калкулатора који допушта и задавање одговарајуће вредности промењљивима (именима) уз поштовање претходно наведеним правила.

2.3 Типови података

Осим бројева (целих, у покретном зарезу и других који нису обухваћени јер би иначе обим рада постао превише велики) и променљивих у Python-у постоје и типови података. То су:

- Ниске (енгл. *strings*)
- Уређене n-торке (енгл. *tuples*)
- Листе (енгл. *lists*)
- Мапе (енгл. *dictionary*)

Ови типови података могу бити мутабилни (променљиви) и немутабилни (непроменљиви). Тако су листе и речници мутабилни, док су уређене n-торке и ниске непроменљиви. То значи да се, на пример листа која је задата једној променљивој може трајно променити коришћењем наредби и поступака које ће касније бити описане, док рецимо уређене n-торке не могу да се мењају већ само може да се користи њихов садржај.

2.3.1 Ниске

Ниска је заправо текст, односно може се размишљати о нисци као секвенци слова. Тако сва слова, бројеви и специјални знаци из овог рада, могу чинити једну ниску. Да би се креирала ниска потребно је поставити низ знакова, тј текст у оквиру наводника. Ниска обично бива додељена некој променљивој.

```
>>> ovo_je_niska = "Hello World!"
```

Кôд 5: Креирање ниске

Наводници могу бити двоструки и једноструки, резултат ће остати исти. Ако постоји потреба да двоструки или једноструки наводници буду део ниске, иска ће бити уоквирена једноструким или двоструким наводницима респективно:

```
>>> ovo_je_niska = "Hello World!"
>>> i_ovo_je_niska = 'Hello World!'
>>> i_ovo = 'Kako se kaze "String" na srpskom jeziku?'
>>> moze_i_ovo = "Python je nastao u '90-im godinama proslog veka"
```

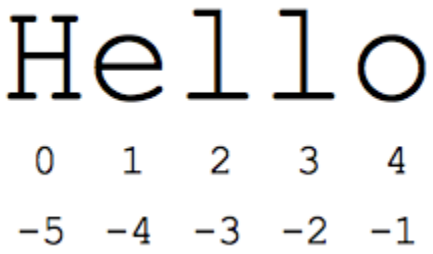
Кôд 6: Примери креирања ниски

Приликом приказа на излаз, у ниску могу бити убачени разни динамички или статички подаци који представљају вредност променљиве. Таква радња се обавља помоћу знака %s у оквиру ниске:

```
>>> moj_prosek = 7.6
>>> poruka = 'Moj prosek je bio %s, a mogao je biti i bolji...'
>>> print (poruka % moj_prosek)
Moj prosek je bio 7.6, a mogao je biti i bolji...
```

Кôд 7: Убацивање података у ниску

Будући да је ниска заправо низ знакова, сваки знак има свој редни број у ниски. Први знак је на нултом месту, док се последњем знаку приступа са -1. Сад кад је познато како се може



Слика 4: редни бројеви карактера у ниски

приступити појединим знаковима у ниски, могу се набројати које операције са нискама имамо. Постоји више операција са нискама. Табела која следи приказује операције са нискама:

ОПЕРАЦИЈЕ СА НИСКАМА		
Операције	**Објашњење**	**Пример**
+	спајање ниски - конкатенација	s + t
*	множење, одн. мултипликација ниски	s * 3
s[i]	враћа i-ти знак у ниски	s[-1]
s[n:m]	враћа све знаке ниске између позиција n и m	s[2:4]
len(s)	враћа целобројну дужину ниске	len("Hello!")
str(n)	претвара број у ниску	str(23)

Табела 1: Операције са нискама

Посебно је интересантна операција исецања ниске (енгл. *slice*), чиме се даје могућност да се од ниске исече и одвоји тачно онај део који је потребан.

```
>>> s = "Hello"          # niska s pokazuje na niz slova Hello
>>> print (s[1:4])       # svi znakovi pocev od drugog zakljucno sa cetvrtim
ell
>>> print (s[1:])        # svi znakovi pocev od drugog do kraja niske
ello
>>> print (s[:])         # svi znakovi od pocetka do kraja
Hello
>>> print (s[1:100])     # svi znakovi pocev od drugog zakljucno sa stotim
ello                     # ako ne postoji 100-i znak, vraca se kraj niske
>>> print (s[-1])        # odredjuje se poslednji znak niske
o
>>> print (s[-4])        # odredjuje se 4. znak od pozadi
e
>>> print (s[:-3])       # svi znaci pocev od pocetka pa do treceg od pozadi
He
>>> print (s[-3:])       # svi znaci pocev od treceg od pozadi do kraja niske
llo
```

Kôд 8: Комадање ниске

2.3.2 Листе

Листа је мутабилна структура података, у којој се може сачувати секвенца произвољних елемената. Листу могу чинити бројеви, ниске или бројеви и ниске у једној листи. Листа може да садржи друге листе.

Листа се означава помоћу угластих заграда између којих се набрајају чланови листе, одвојени запетом. Ако између две угласте заграде нема ништа, креирана је празна листа. Листа се може мењати, могу се избацивати одговарајући чланови листе, може се променити садржај одговарајућих чланова, а листа се може и допунити новим члановима. Све ове особине чине листе врло корисним оруђем у програмирању.

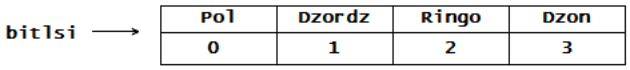
```
>>> bitlsi = ['Pol', 'Dzordz', 'Ringo', 'Dzon']
>>> print(bitlsi)
['Pol', 'Dzordz', 'Ringo', 'Dzon']
```

Kôд 9: Креирање листе

```
>>> prazna_lista = []
>>> print(prazna_lista)
[]
```

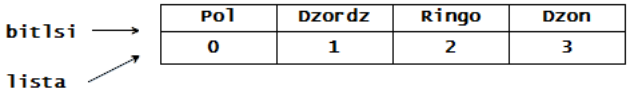
Kôд 10: Креирање празне листе

Кад се креира листа под неким именом, онда то име (променљива) реферише на део меморије у који је уписана дата листа.



Слика 5: Креирање листе

Ако је потребно да се креира нова листа (на пример *lista*) и њој додели вредност листе *bitlsi*, тада ће и нова листа референсати на исти меморијски простор. Ако изменимо садржај листе *lista*, аутоматски се мења и садржај листе *bitlsi*.



Слика 6: lista = bitlsi

На сликама 5 и 6, се може уочити редни број чланова листе формира на исти начин као и код ниски, дакле: први члан је под редним бројем 0, следећи је 1, итд. Ако је потребно обележити последњи члан листе, користи се редни број -1, претпоследњи је -2, итд.

На листе је могуће применити и исецање листи, на сличан начин како је описано при опису операција са нискама.

```
>>> bitlsi = ['Pol', 'Dzordz', 'Ringo', 'Dzon']
>>> print (bitlsi[0:2])
['Pol', 'Dzordz']
>>> print (bitlsi[1])
Dzordz
>>> print (bitlsi[-1])
Dzon
```

Кôд 11: Исецање листи

Оно што највише разликује листе од ниски је способност мутације. Листама се могу додавати или избацивати чланови, листе се могу сортирати. У наредној табели набројане су операције са листама.

ОПЕРАЦИЈЕ СА ЛИСТАМА		
Операције	**Објашњење**	**Пример**
list[i]	враћа i-ти члан листе	bitlsi[0]
list[m:n]	даје све чланове почев од m-тог па закључно са n-1	bitlsi[0:2]
len(list)	враћа број чланова листе	len(bitlsi)
+	спаја две листе	bitlsi + stonsi
*	мултипликује садржај листе	bitlis * 2
МЕТОДЕ		
append	додаје нови члан листе на крај листе	bitlsi.append('Pete')
insert	додаје нови члан на датом месту у листи	bitlsi.insert(1, 'Brian')
index	враћа редни број датог члана	bitlsi.index('Pol')
remove	брише дати члан листе	bitlsi.remove('Dzon')
pop	враћа и брише последњи члан листе	bitlsi.pop()
sort	сортира листу	bitlsi.sort()

Табела 2: Операције и методе са листама

```
>>> bitlsi = ['Pol', 'Dzordz', 'Ringo', 'Dzon']
>>> bitlsi.append('Pit')
>>> print(bitlsi)
['Pol', 'Dzordz', 'Ringo', 'Dzon', 'Pit']
>>> bitlsi.remove('Pit')
>>> print(bitlsi)
['Pol', 'Dzordz', 'Ringo', 'Dzon']
>>> bitlsi.insert(1, 'Pit')
>>> print(bitlsi)
['Pol', 'Pit', 'Dzordz', 'Ringo', 'Dzon']
>>> bitlsi.pop()
'Dzon'
>>> print(bitlsi)
['Pol', 'Pit', 'Dzordz', 'Ringo']
>>> bitlsi.sort()
>>> print(bitlsi)
['Dzordz', 'Pit', 'Pol', 'Ringo']
```

Кôд 12: Мењање листе

Листе су моћно средство програмирања у Python-у, али потребно је бити опрезан кад се употребљавају одговарајуће методе, како се не би јавила грешка(нпр. покушај да се избаци елемент који не постоји у датој листи, доводи до грешке). Такође, треба бити опрезан и код мутирања листе, јер као што је показано ако две променљиве реферишу на исту листу, мењање једне, проузрукује и мењање друге.

2.3.3 Мапе

Мапе или речници(енгл. *dictionaries*) представљају колекцију елемената, слично као што је то случај и са листама. Разлика између мапа и листа је у томе што се код мапа уместо редних бројева чланова листа, користе парови кључева и њима одговарајућих вредности. Дакле, мапа је структура која се записује тако што се између витичастих заграда набрајају парови који чине један кључ и једна или више вредности везаних за тај кључ. Вредности могу бити бројеви, ниске, листе, уређене n-торке.

```
>>> bitl_list = [['Pol', 'bas', 'vokal'],
                 ['Dzordz', 'gitara'],
                 ['Ringo', 'bubanj'],
                 ['Dzon', 'gitara', 'vokal']]
>>> print (bitl_list)
[['Pol', 'bas', 'vokal'], ['Dzordz', 'gitara'], ['Ringo', 'bubanj'],
['Dzon', 'gitara', 'vokal']]
>>> bitl_map = {'Pol' : ['bas', 'vokal'],
                'Dzordz' : ['gitara'],
                'Ringo' : ['bubanj'],
                'Dzon' : ['gitara', 'vokal']}
>>> print(bitl_map)
{'Pol': ['bas', 'vokal'], 'Dzordz': ['gitara'],
'Ringo': ['bubanj'], 'Dzon': ['gitara', 'vokal']}
```

Кôд 13: Разлика између листе и мапе

Ако је потребно да се прикаже шта је Пол свирао, уколико су подаци о групи запамћени као листа, потребно је знати који је Пол редни број у листи, док код мапе редни број не постоји и вредностима се приступа помоћу кључа.

```
>>> print (bitl_map['Pol'])
['bas', 'vokal']
```

Кôд 14: Кључ и вредност

Ако је задатак да се утврди ко је све свирао гитару у Битлсима, потребна је петља, тако да ће тај задатак бити решаван у поглављима која следе, када буду обрађивале петље и модули.

Због природе мапа, мутација је прилично другачија у односу на листе. Да би се додао нови елемент, односно пар кључ-вредност, потребно је уз име мапе ставити нови кључ и доделити му вредност или изменити постојећу. Такође помоћу команде *del*, могуће је обрисати кључ, а тиме и вредност везану за њега.

```
>>> bitl_map['Pit']='bubanj'
>>> bitl_map['Dzordz']=['gitara', 'vokal'] # dodan je vokal Dzordzu
>>> del bitl_map['Pit']
```

Кôд 15: Мутација мапа

Могуће је креирати и празну мапу, једноставно се остави празан простор између две витичасте заграде.

```
>>> prazna_mapa = {}
```

Кôд 16: Креирање празне мапе

Методи и функције за рад са мапама су различити у односу на листе, јер не постоје, на пример *pop* и *append* методе који омогућавају мутацију, већ да би се мењала мапа потребно је знати који елемент се убацује, мења или избацује из дате мапе.

МЕТОДИ И ФУНКЦИЈЕ ЗА РАД СА МАПАМА		
Метода	**Објашњење**	**Пример**
map[key]=value	Додељивање или мењање вредности кључа	bitlsi['Dzon'] = 'vokal'
del map[kljuc]	Брисање кључа и вредности из мапе	del bitlsi['Pol']
keys()	Метод којим се сви кључеви дате мапе враћају као листа	bitlsi.keys()
values()	Метод којим се све вредности дате мапе враћају као листа	bitlsi.values()
items()	Метод који враћа листу уређених парова кључ, вредност	bitlsi.items()

Табела 3: Методе и функције за рад са мапама

2.3.4 Уређене n-торке

У Python-у постоји начин да подаци сместе у уређене n-торке, где $n \in \mathbb{N} \cup \{0\}$. Подаци се смештају између две заграде и раздвојени су зарезом. Python интерпретатору није потребно да их ставимо између заграда, док год су подаци раздвојени зарезом, већ ће он свако набрајање схватити као уређену n-торку, чак и ако нема заграде.

```
>>> fibo = (0, 1, 1, 2, 3)
>>> print (fibo)
(0, 1, 1, 2, 3)

>>> fibo1 =0, 1, 1, 2, 3  # uredjena n-torka bez zagrada
>>> print (fibo1)
(0, 1, 1, 2, 3)

>>> prazan = () # kreiranje prazne uredjene n-torke
>>> print (prazan)
()

>>> jedan_clan = (1,) # jednoclana uredjena n-torka mora imati
>>> print (jedan_clan) # zarez, iako ne postoji drugi broj
(1,)
```

Кôд 17: Креирање уређене n-торке

У уређеним n-торкама је могуће сместити вредности и других типова података, као што су бројеви и ниске. Такође, могуће је сместити и листе и друге уређене n-торке, док год се поштује горе наведена синтакса.

```
>>> niska = ('a', 'b')
>>> print (niska)
('a', 'b')
>>> lst = ([1, 2, 3], [1,2])
>>> print (lst)
([1, 2, 3], [1, 2])
>>> tapl = ((1,), (), (1, 2, 3))
>>> print (tapl)
((1,), (), (1, 2, 3))
```

Кôд 18: Уређене n-торке могу садржати и ниске и листе и друге уређене n-торке

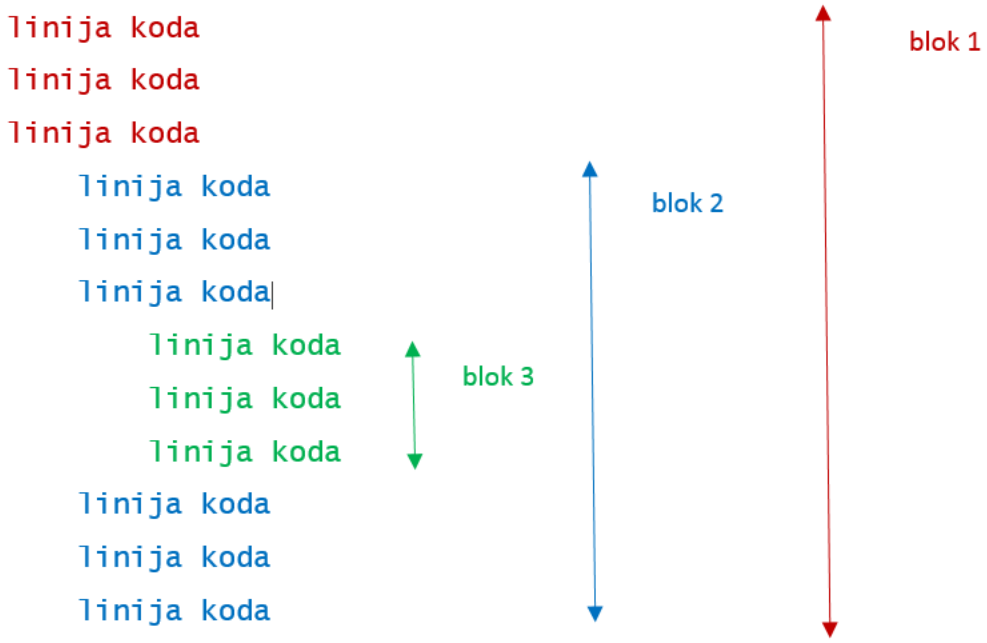
Оно што највише разликује уређене n-торке од осталих типова података је да су они у потпуности непроменљиви. Када се креира уређена n-торка, могуће је искључиво користити њене вредности, док је немогуће мењати или додавати вредности уређених n-торки. Могуће је, уколико се за тим укаже потреба доћи до појединих чланова уређених n-торки, ако се зна њихов редни број, а могуће је исецати уређене n-торке, слично као код ниски.

2.4 Услов

У програмирању се често постављају "да" или "не" питања и доносе се одлуке шта урадити у односу на одговор. На пример, може се поставити питање: "Да ли студирате математику мање од 10 година?" и ако је одговор "Да", то се може протумачити као: "Ви сте сјајан студент!".
Често се комбинују услови(питања) и одговори у оквиру *if* наредбе. Услова може бити више у једном изразу и можемо их комбиновати.

2.4.1 Програмски блок

Линије кода после двотачке морају бити постављене у блок, иначе их Python интерпретатор неће правилно протумачити. Да би блок био разумљив интерпретатору, блок мора бити увучен тачно четири (празна) карактера[3] у односу на линију која је захтевала програмски блок[**?**]. Линије кода се групишу у исти блок морају бити на истој удаљености од маргине[4]. Структура блокова у Python-у би могла да буде оваква:



Слика 7: Пример програмских блокова у Python-у

2.4.2 Наредба if

Следи начин на који би се претходно питање, тј. услов, могао реализовати:

```
>>> godine_studiranja = 6
>>> if godine_studiranja < 10:
```

[3]Празни карактери се на енглеском називају белине(енгл. *Whitespaces*
[4]*indent, енгл.*

```
        print('Vi ste sjajan student')
```

Кôд 19: Пример услова

2.4.3 Наредба if-then-else

Могуће је проширити дефиницију *if* наредбе како би, на пример у претдном примеру пронашло колико је ко година студирао. За такву структуру је потребна кључна реч *else*.
Укратко, ако је услов у *if* делу тачан, извршава се блок испод те линије, а ако није тачан, извршава се блок испод *else* линије ако та линија кода постоји. Ако не постоји, а услов није тачан, неће се ништа извршити, већ интепретатор наставља извршавање даљих линија кôда.

```
>>> if godine_studiranja < 10:
        print('Vi ste sjajan student')
    else:
        print('Nije strasno')
```

Кôд 20: Пример за наредбе IF - ELSE

2.4.4 Услови и комбиновање услова

Упоређивање две величине даје једну од две вредности: тачно или нетачно. Следећи оператори се користе за упоређивање:

Оператор	**Дефиниција**
==	једнако
!=	није једнако
>	веће од
<	мање од
>=	није мање од
<=	није веће од

Табела 4: Оператори упоређивања

На пример, ако је потребно проверити да ли нека особа има 20 година, користи се оператор једнакости: *if godine==20:*

Услови се комбинују коришћењем логичких оператора *и* и *или*. Оператор конјункције је *and*, а дисјункције *or*. Тако да на пример можемо овако рећи: *if godine > 18 and bodovi >=50:* и слично. Такође, у услову се може испититати да ли је или није неки елемент члан листе,

уређене n-торке, ниске или мапе. За то се користе оператори *in* и *not in*. На пример: *if elem not in lista:*.

2.5 Петље

Програмери не воле много да се понављају, штавише у Python-овом "правилнику" за писање кôда, PEP[**?**], постоји правило *DRY*, односно **Don't Repeat Yourself**[5], но понекад је потребно да интерпретатор одређени блок поновљено извршава коначно много пута. Такав део програма се назива петља. Постоји више врста петљи у Python-у, овде ће бити описане две.

2.5.1 Петља while

Као што дословни превод са енглеског каже, *док* је одређени услов испуњен, исти кодни блок се врти у петљи. Ако услов није тачан, петља се прекида и извршавање програма се наставља иза петље. Услови петље се дефинишу на претходно описани начин. На пример:

```
>>>i=1
>>>while  i<=10:
        print(i)
        i=i+1
```

Кôд 21: Пример while петље

Ако је потребно да се изађе из петље пре него што услов постане нетачан, користи се кључна реч *break*. У том случају петља се прекида и програм наставља даље да се извршава од прве наредбе која следи иза петље.

[5] енгл., *не понављај се*

2.5.2 Петља for

Ова петља се користи на сличан начин као и петља *while*, с тиме да се користи нова кључна реч - *range*. На пример:

```
>>>for i in range(1, 11):
      print(i) // stampaju se brojevi od 1 do 10
>>>for i in list:
      print(list[i]) //stampaju se elementi liste
```

Кôд 22: Примери *for* петље

И *for* и *while* петље се могу угњежђавати и то вишеструко са више петљи.

2.6 Функције и модули

Ако се довољно често употребљава одређени кодни блок, који би могао поново да се искористи, онда је пожељно правити функцију. Организација кôда у којој се крупне целине деле на мање и смештају у функције је пожељна, јер омогућава бољу прегледност кôда, а тиме и лакше проналажење грешака. Такође, у том случају је могуће лакше да мењати кôд, прилагођавати га другим апликацијама, итд.

2.6.1 Функције у Python-у

Да би се употребила функција *func*, која има улазне параметре, на пример $a_1, a_2, \ldots, a_n$, она мора да се позове у кôду са:

```
>>>func(a1, a2, ..., an)
```

Дефинисање функције почиње са кључном речи *def*, иза које се наводи назив и параметри функције. Функција не мора имати параметре, а ако их има, онда их има коначно много. На крају реда се поставља специјални знак - двотачка, која у Python-у има улогу да упозори интерпретатор да иза ње следи кодни блок. У следећи ред кôда, увученом за 4 празна карактера, започиње кодни блок дате функције.

```
>>>def min(a, b):
       if a < b:
           return a
       else:
           return b
```

Кôд 23: Дефинисање функције

Да би функција вратила неку вредност потребно је да у дефиницији функције постоји кључна реч *return* после које се наводи вредност која ће бити резултат функције.

2.6.2 Модули

Модули служе за груписање функција, променљивих и других структура у веће и моћније програме. Неки модули су уграђени у сам Python: на пример, *tkinter*[6]. Такође, постоји могућност учитавања модула које су други поставили на интернет, а који помажу лакшем и бржем писању жељеног кôда. Неки од познатијих таквих модула су *PIL*[7] (Python Imaging Library), Panda3D, итд.
Да би се користио модул, потребно је на почетку програма употребити кључну реч *import*, после које се наводи име модула. На пример: *import time*.

[6] *tkinter* служи за прављење игара у Python-у
[7] PIL више није актуелан од верзије Pythona 2.6. Наследник овог модула је *PILLOW*

2.7 Класе и објекти

Модеран програмски језик не може се замислити без подршке објектно-ориијентисаном програмирању[8]. Python, наравно, омогућава коришћење објектно-оријентисаног програмирања и даје му пуну подршку.

2.7.1 Објектно-оријентисано програмирање

Овај рад нема амбицију да се бави Објектно-оријентисаним програмињем(у даљем тексту: ООП). У даљем тексту ће се у краћим цртама навести значај, особине и предности ООП.
Појам Објектно-оријентисаног програмирања први пут се помиње почетком '70-их година прошлог века, приликом представљања програмског језика *Smalltalk* (први језик који је у себи имао елементе ООП је био *Simula*67). Касније се највише везује за програмски језик Java, да би се данас широко примењивао у програмирању.
Објектно-оријентисано програмирање је засновано на концепту објекта. *Објекти* су структуре података са придруженим скупом процедура и функција које се називају *методи* и служе за рад са подацима који припадају објекту. Уобичајено је да су методи једини начин за рад са објектима. Сваки објекат је примерак или инстанца неке *класе*. У класи се дефинише садржај објеката те класе и скуп метода који ће омогућити рад са објектима.
Готово у свим објектно-оријентисаним програмским језицима постоји *наслеђивање* као механизам за креирање нових класа из већ постојећих. На тај начин се добијају натклассе и потклассе. Уобичајена је могућност у објектно-оријентисаном програмирању дефинисање и потклаcа класе, која је у односу на своју супер-класу *дете*, док је супер-класа *родитељ*. Потклаcа наслеђује све садржаје објеката класе и методе од натклассе. Типичан пример за овакав начин наслеђивања је класа Животиња, где можемо дефинисати потклассе Сисари, Инсекти, Птице, Рибе, итд. Такође, можемо дефинисати и класу Торбари, која ће бити потклаcа Сисара и тај процес наслеђивања можемо продужити коначно много.

2.7.2 Класе и објекти у Python-у

Да би се креирала класа у Python-у, потребно је користити кључну реч *class*, после које се наводи име класе и евентуално класу натклассе, које је та класа наследила. На пример:

```
>>>class Zivotinje:
       pass
>>>class Sisari(Zivotinje):
       pass
```

Кôд 24: Дефинисање класа

Кључна реч *pass* значи да та класа не садржи никакве особине објеката нити методе.
Када постоји потреба за увођењем конкретног примерка класе, тј. објекта неке класе, тада је

[8] краће *ООП*

потребно само да навести име објекта и после знака једнакости навести класу и особине које припадају датом објекту, као у следећем примеру:

```
>>>class Macke(Sisari):
       pass
>>>tosa = Macke()
```

Кôд 25: Креирање објекта

У Python-у не постоје конструктори и деструктори као у неким другим објектно-оријентисаним програмским језицима. Довољно је навести име класе и инстанца је препозната код интерпретатора.

2.7.3 Методи

У класи је могуће креирати скуп функција. Такве функције се називају методима дате класе. Креирање метода се врши дефинисањем функције унутар класе.
Објекту се додељује метод, тако што се после имена објекта поставља тачка и наводи име метода са аргументима, набројаним унутар заграде. Метод не мора да има аргументе.

```
>>>class Macke(Sisari):
       def predenje():
           print('purrrrr')
>>>tosa.predenje()
>>>purrrrr
```

Кôд 26: Методи класе

3 Претрага Интернета и обрада хипервеза

3.1 Проналажење информација

Проналажење информација може се дефинисати као процес претраживања унутар неког документа и/или колекције докумената за потребном информацијом.[?]. Кроз историју се проблем проналажења информација у документима почео решавати још у 5. веку п.н.е. у Античкој Грчкој, кад је први пут уведен *садржај* у неки од свитака папируса, пошто књиге у данашњем облику тада још нису постојале. У Старом Риму проблем су решавали етикетама налик на данашње *Post-it* налепнице. Све се, наравно, променило са проналаском штампарске машине, средином 15. века. Крајем 19.века почиње да се имплементира Дјуијев децимални систем[9] , каталог картица. У 20. веку долази до употребе технологије у претраживању докумената, па се тако појављују микрофилмови. У '60-им годинама прошлог века направљена је прва машина која је радила претрагу информација, а ради се о *MARC*[10] рачунару. Данас се у традиционалном претраживању и проналажењу информација користи систем картица (на пример у библиотекама), али и компјутерски помогнути системи. У овим последњим постоје три карактеристична модела за претраживање: *буловски*, *векторски*, *модели вероватноће*[?, Ch 1.2]. Постоји на хиљаде модела, али су сви настали као варијанте три горе наведена.

Буловски модел користи систем егзактног подударања да нађе документ који корисник захтева. Име је добио по Буловој алгебри, чије логичке операторе користи. Не постоји концепт парцијалног подударања што је велики проблем, ако је потребно да пронађи документ, за који се не зна тачан назив. Проблеми претраживања информација, као што су проблем синонима и проблем вишезначности, овде се не могу избећи.

Векторски модел је увео Џералд Салтон,'60-их година прошлог века. Базира се на трансформацији текстуалних података у нумеричке векторе и матрице и примењује се матрична анализа за откривање повезаности кључних речи. Овакви модели решавају проблеме синонима и вишезначности. Резултати се могу презентовати сортирани према степену релевантности.[?]

Модели вероватноће покушавају да процене вероватноћу којом ће корисник наћи жељени документ у односу на упит. Резултати се могу поређати по изгледима релевантности. Није једноставно имплементирати овакве моделе у рачунарско окружење.

3.2 Проналажење информација на вебу

Када је у марту 1989. године британски инжењер Тим Бернерс-Ли послао предлог[11] да се унапреди информациони систем у *CERN*-у[12], није ни слутио у шта ће се претворити једноставна потреба за хипервезивањем основних података о запосленим у *CERN*-овом информационом систему[?]. Тим Бернерс-Ли је за потребе новог информационог система измислио нови језик који је назвао *Hypertext Markup Language*, одн. **HTML**. Такође, осмислио је и читач HTML, кôда који је назвао *World Wide Web* и он се у исто време користио и за прављење веб страница.

Од прве веб странице из маја 1990. године до данас је прошло више од 20 година, а у овом тренутку постоји неколико милијарди веб страница. Јасна је мотивација која се од почетка

[9] Dewey, 1872. Сортирао је колекције према тематици, на пример, троцифреним бројевима који почињу са 1 је означавао филозофију, са 2 религију, итд. Тако да је, на пример, књига са идентификационим бројем 142 означавала књигу о филозофији.

[10] енгл., *Machine Reading Catalog*

[11] `http://www.w3.org/History/1989/proposal.html`

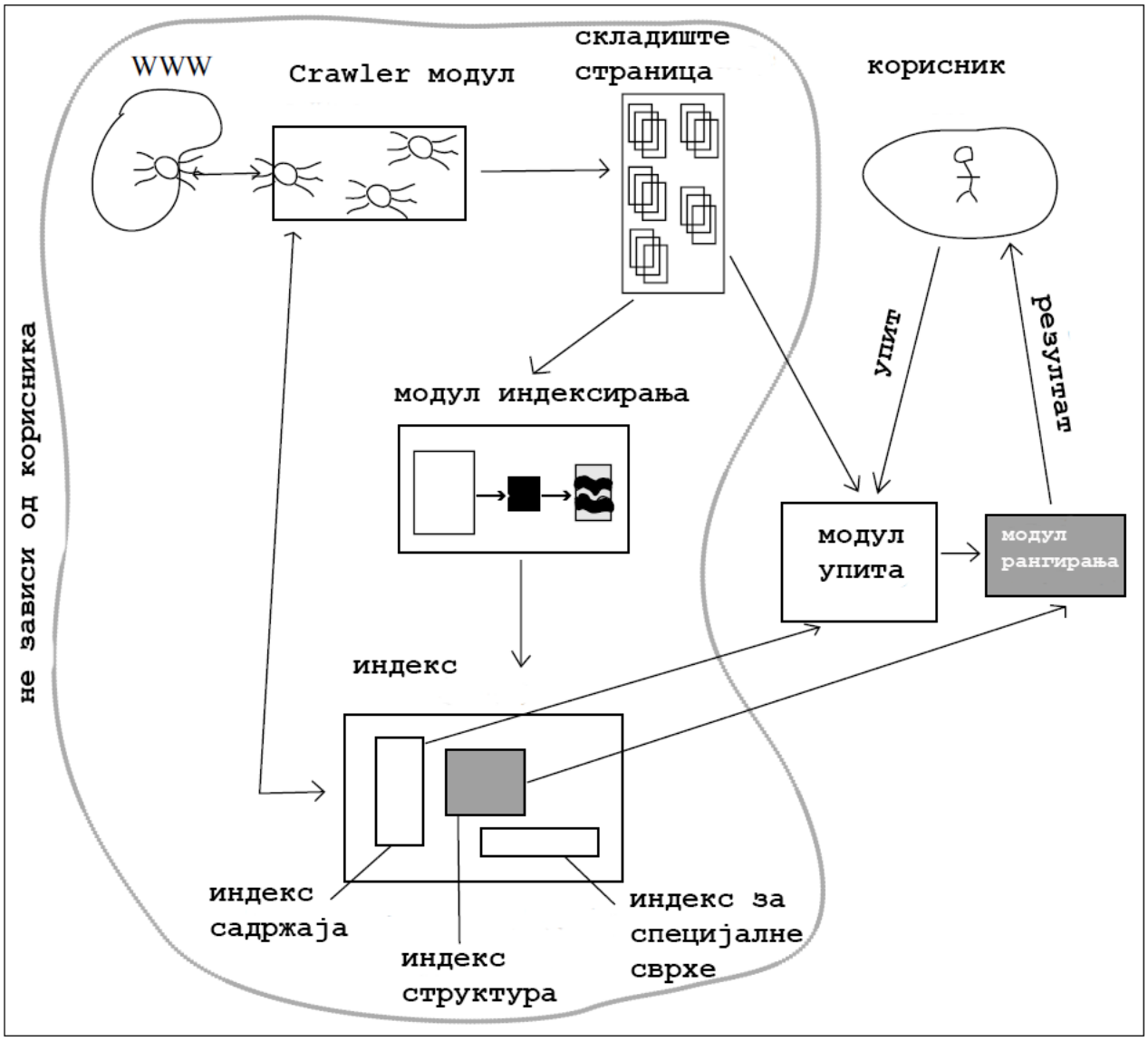
[12] Европска организација за нуклеарна истраживања (франц., *Conseil Européen pour la Recherche Nucléaire*)

јавила корисницима веба, а то је како наћи праву информацију. Веб је огроман, динамичан, самоорганизован и хиперповезан.[?, Ch 1.3.1] Свако може поставити веб страницу. Странице се мењају дневно, па чак и чешће. Нове странице се појављују сваки минут. То су реални проблеми веб претраживања, на које није могло да одговоро традиционално претраживање информација. Такође, корисници ретко погледају више од 10 до 20 докумената који су понуђени, што појачава потребу да резултати морају бити јако прецизни и брзо достављени кориснику.
Први Интернет претраживач је био *Archie*, направљен 1990. године, од стране канадског студента Алана Емтејџа. Примарни протокол је био *ftp*. Први робот који је аутоматски индексирао странице звао се *World Wide Web Wanderer* који је своје резултате стављао у прву базу хипервеза веб страница *Wandex*. До средине '90-их направљено је на хиљаде *веб-паукова* тј. робота који су складиштили информације на вебу. На жалост, нису знали како да искористе те информације на добар начин. 1994. године појавио се *Yahoo* као збир неколико популарних страница и са директоријумом популарних страница. У том тренутку *Yahoo* се не може сматрати претраживачем. Али 1994. године се појављују два значајна *веб-паука*-а, *WebCrawler* и *Lycos*, чији се број сакупљених хипервеза мерио у десетинама милиона. У децембру 1995. године појављује се први прави веб претраживач *AltaVista* који је омогућавао примену логичких оператора. *AltaVista* је имала врло лош алгоритам рангирања страница, који се заснивао на провери тзв. мета-података из заглавља HTML документа и врло лако је могао да се злоупотреби и зато је пројекат пропао крајем 20. века, а са појавом *Google*-а. Два су кључна алгоритма за даљи развој веб претраживања: *PageRank* и *HITS*. Први је омогућио Гуглу да постане оно што данас јесте, а то је најкоришћенији претраживач са највише индексираних страница, док је *HITS* дело Џона Клајнберга из IBM-а купљен од стране *Teome*, данашњег `ask.com`. Оба алгоритма су се јавила независно један од другог исте, 1998. године. Више о PageRank алгоритму у даљем делу рада. Данас имамо много интернет претраживача, попут Google-a, Bing-a, Yahoo-a, итд.

3.3 Процес веб претраживања

Веб претраживач је веб апликација која ће претражити веб странице, извући све могуће хиперлинкове, рангирати их и на крају дати најбоље могуће решење за задату кључну реч. Апликација се састоји из више елемената:

- модул веб-паука
- модул индексизације
- индекс
  - индекс садржаја
  - индекс структуре
  - индекс за специјалне сврхе
- складиште страница
- модул упита
- модул рангирања



Слика 8: Процес веб претраживања [**?**, Ch 1.3.2]

Процес се састоји од узајамног деловања претходно набројаних модула. Тако веб-паук скупља странице по вебу, целе странице одлаже у *складиште веб страница*, док хипервезе шаље у индекс, који их поставља уз кључне речи које добија из модула индексовања. Кад корисник пошаље упит, модул упита тражи од индекса скуп свих решења, која се кроз модул рангирања на крају постављају као резултати корисниковог упита. Модел процеса може се представити као што је то урађено на слици 8.

3.4 Извлачење хипервезе

У процесу реализације кôда веб претраживача, прво је потребно написати програм веб-паука, који ће претражити страницу и наћи прво један, па онда и све остале хипервезе. Да би се пронашла хипервеза, потребно је знати како изгледа веб страница.

3.4.1 Структура HTML странице

Веб страница је заправо само дугачак низ карактера образованих у HTML кôд. Веб прегледач претвара кôд у изглед који се обично добија на екрану приликом избора неке странице. Да би се видео дати кôд који генерише веб страницу, зависно од веб прегледача који се користи (*Internet Explorer*, *Firefox*, *Safari* или *Chrome*, на пример), потребно је потражити опцију "View Page Source", "View Source" или слично. Структура овог рада не омогућава претерано улажење у HTML, те ће се упрошћено навести како изгледа структура HTML кôода[13]. На пример:

- <html> таг;
- Заглавље, у којем се налазе подаци о наслову странице, аутору, разним мета подацима везаним за страницу и све то се налази између <head> и </head> ознака;
- Садржај странице, који се исписује између <body> ознака;
- затвара се </html> ознаком

Интересантно за овај рад је како су означене хипервезе у том кôду.

3.4.2 HTML <a> tag

За веб-паука од највеће важности је пронаћи хипервезе ка другим веб страницама. Хипервезе у HTML кôду се налазе унутар <a> ознаке. Наиме, хипервеза се задаје на следећи начин:

```
<a href="<url>">
<a href="http://www.matf.bg.ac.rs">
```

Кôд 27: <a> ознака

Дакле, да би се пронашла хипервеза унутар HTML кôда, потребно је прво пронаћи <a> ознака.

3.4.3 Налажење првог хиперлинка

Алгоритам за налажење прве хипервезе у HTML кôду, почиње од налажења <a href=" дела, а сама хипервеза ће се наћи између знакова навода који следе после горе поменуто дела кôда. Следи кôд који то успешно ради:

```
1  # u promenljivoj page je smestena niska celog
2  # HTML koda internet stranice
3  page = '<sadrzaj veb stranice>'
4
5  # trazi se prvo pojavljivanje <a> oznake i
6  # smesta se u promenljivu start_link
7  start_link = page.find('<a href=')
8
9  # trazi se prvo pojavljivanje znaka navoda
10 start_quote = page.find('"', start_link)
11 # trazi se zatvaranje navodnika
```

[13] више о томе на следећој страници `http://www.w3.org/TR/html401/struct/global.html`

```
12 end_quote = page.find('"', start_link+1)
13 # hiperveza je sve izmedju dva znaka navoda
14 url = page[start_quote + 1 : end_quote]
```

Кôд 28: Налажење првог хиперлинка

3.5 Налажење свих хипервеза на страници

У претходном поглављу је дат кôд за налажење једне хипервезе унутар HTML кôда. Да би се пронашле и остале хипервезе потребно је да наставити са процесом тражења следеће хипервезе. Ради тога, уводи се процедура која ће омогућити налажење следеће хипервезе.

3.5.1 Процедура "get_next_target"

Да би се пронашла прва следећа хипервеза, односно да би се установило где ће бити почетак следеће хипервезе, потребно је пронаћи следећу <a> ознаку и тада поновити кôд за налажење прве хипервезе. Та процедура би требало да изгледа овако:

```
1 def get_next_target (page):
2     start_link = page.find('<a href=')
3     start_quote = page.find('"', start_link)
4     end_quote = page.find('"', start_quote+1)
5     url = page[start_quote+1:end_quote]
6     return url, end_quote
```

Кôд 29: Процедура налажења прве следеће хипервезе

Ова процедура враћа вредност хипервезе коју је пронашла, али и позицију завршног знака навода, како би се знало где је стала процедура и наставило са тражењем осталих хипервеза.

3.5.2 Ако нема линкова?

Потребно је решити проблем са горе поменутом процедуром у ситуацији да нема хипервеза у страници. Решење је да се постави услов, ако се не нађе почетак хипервезе, онда се враћа кључна реч None, која означава празну ниску.

```
1 def get_next_target (page):
2     start_link = page.find('<a href=')
3     if start_link == -1:
4         return None, 0
5     start_quote = page.find('"', start_link)
6     end_quote = page.find('"', start_quote+1)
7     url = page[start_quote+1:end_quote]
8     return url, end_quote
```

Кôд 30: Испитивање да ли страница садржи хипервезу

За испитивање у услову је коришћена вредност -1, јер Python враћа ту вредност ако не нађе задату ниску, тј. ако ниска не постоји.

3.5.3 Све хипервезе

Сад је могуће, користећи процедуру get_next_target(в. кôд 30), одштампати све хипервезе једне веб странице. Претпоставља се да је у променљивој *page*, смештен HTML кôд неке веб странице.

```
1 def print_all_pages(page):
2     while True:
3         url, endpos = get_next_target(page)
4         if url:
5             print url
6             page = page[endpos:]
7         else:
8             break
```

Кôд 31: Процедура штампања свих хипервеза

У претходном коду користи се "while True", пошто је потребно налазити нове хипервезе све док их има на страници, а ако их нема, наредбом **break** излази се из петље.

4 Креирање "веб-паука"

4.1 Смештање хипервеза у листу

Користећи претходно описано проналажење свих хипервеза(в. кôд 31), а за потребе креирања веб-паука, потребно је сакупити све хипервезе у неку колекцију, како би касније ти подаци могли бити даље процесуирани.

У наредном кôду, све хипервезе ће бити скупљене у листу. Уместо листе, могуће је користити и друге структуре као на пример мапе, али такви примери ће бити реализовани касније у оквиру овог рада.

```
1  def get_all_links(page):
2      links = []
3      while True:
4          url, endpos = get_next_target(page)
5          if url:
6              links.append(url)
7              page=page[endpos:]
8          else:
9              break
10     return links
```

Кôд 32: Процедура смештања свих хиперлинкова у листу

У линији 4 се користе уређене n-торке, додељују се вредности променљивима **url** и **endpos**. Ако хипервеза постоји, биће додата у листу и траже се друге хипервезе почев од последњег знака наводника претходне хипервезе. Иначе, ако линк не постоји, петља се прекида и процедура враћа листу свих хипервеза са дате веб странице.

4.2 Завршетак веб-паука

Веб паук би требао да нађе све хипервезе на задатој веб страници и да их смести у листу. Даље, веб-паук наставља процес следећи пронађене хипервезе и на тим веб страницама ће налазити нове хипервезе. Да се процес тражења хипервеза не би више пута понављао на истим веб страницама потребно је користити две променљиве:

tocrawl у овој листи ће бити смештене странице које је потребно прегледати

crawled у овој листи ће бити смештене странице које су већ прегледане

4.2.1 Завршни кôд

Претпоставља се да су познате процедуре **get_next_target** и **get_all_pages**, чији кôд је наведен у претходним поглављима(в. кôдове 30, **??**). Да би се кôд нове процедуре **crawl_web** учинио прегледнијим, биће уведена и процедура **union** која прави унију две листе. Процедура **get_page** узима хипервезу и враћа HTML кôд те стране. Ако се страница не може отворити из разних разлога и/или је дата веб страница празна, процедура враћа празну ниску.

```
1  def get_page(url):
2      try:
3          import urllib
```

```
4           return urllib.urlopen(url).read()
5       except:
6           return ""
7
8   def union(a, b):
9       for e in b:
10          if e not in p:
11              p.append(e)
12
13  def crawl_web(seed):
14      tocrawl = [seed]
15      crawled = []
16      while tocrawl:
17          page = tocrawl.pop()
18          if page not in crawled:
19              union(tocrawl, get_all_links(get_page(page)))
20              crawled.append(page)
21      return crawled
```

Кôд 33: Веб паук

Процедура прво поставља две листе, оно што је потребно претражити и ту поставља први елемент, иницијалну страницу. Друга променљива је она у коју се смештају странице које су већ претражене и она је иницијално празна. Кад се страница претражи, она се додаје у **crawled** листу да се не би више пута непотребно претраживала.
У линији 9 почиње петља, која ради док листа **tocrawl** има елемената. Кад листа више нема страница за скенирање, процедура враћа листу прегледаних страна. У петљи се скида последња хипервеза из **tocrawl** листе и на основу одговора после питања из услова да ли је та страница већ претражена, процедура је даље претражује или не. На крају хипервеза бива додата листи већ претражених страна.

4.2.2 Ограничен број страна по свакој хипервези

Уколико не постоји потреба за претраживањем свих веб страница на које упућују хипервезе које су прегледане у току процесу тражења свих хипервеза који је описан у претходном поглављу , већ је потребно прегледати само првих неколико страница, тада да је могуће одредити колико ће се веб страница претражити. У наредном примеру, вредност максималног броја прегледаних страница ће бити смештена у променљиву **max_pages**:

```
1   def crawl_web(seed, max_pages):
2       tocrawl = [seed]
3       crawled =[]
4       count = 0 # vrednost brojaca inicijalno na 0
5       while tocrawl and count < max_pages:
6           page = tocrawl.pop()
7           if page not in crawled:
8               union(tocrawl, get_all_links(get_page(page)))
9               crawled.append(page)
10              count = count + 1 # iteracija brojaca posle svakog linka
```

```
11      return crawled
```

Кôд 34: Претраживање са ограниченим бројем страна

Дакле, петља se прекида ако је листа празна или ако је прекорачен максимални број хипервеза по страници.

4.2.3 Ограничена "дубина" претраживања

Ефикаснији начин ограничавања рада веб-паука да не претражује све странице је да се постави одговарајућа "дубина" претраживања. Дубина ће у наредном примеру бити одређена промењливом **max_depth**.

```
1  def crawl_web (seed, max_depth):
2      tocrawl = [seed, 0] # postavlja se 0 kao pocetna dubina
3      crawled = []
4      while tocrawl:
5          layer = tocrawl.pop()
6          depth = layer[1]
7          if depth <= max_depth:
8              for url in layer[0]:
9                  if url not in crawled:
10                     union(tocrawl, [get_all_links(get_page(url)),depth+1])
11                     # dubina se povecala za 1
12                     crawled.append(url)
13     return crawled
```

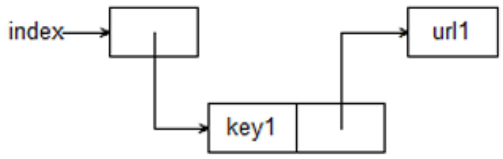
Кôд 35: Скенирање ограничено по дубини

За потребе оваквог ограничавања, измењена је листа **tocrawl**, у којој сада постоји пар елемената: страница и број. Број се повећава кад се прегледају све хипервезе са одређене странице и пређе на нову страну. Тако се ограничава веб-паук да после одређеног броја прегледаних страница заустави свој рад.

5 Добијање одговора на задати упит

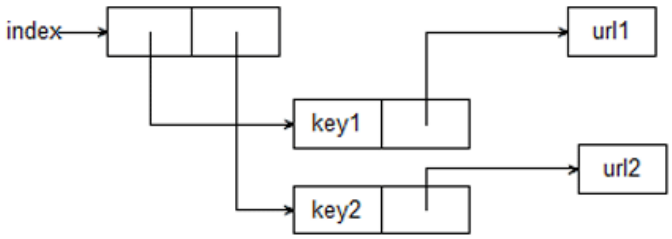
5.1 Одговори на упите

Претраживач би имао мало или нимало смисла ако корисник не би могао да уноси кључне речи за претраживање и да на основу њих добије одговарајуће хипервезе. Зато је поред хипервеза потребно имати и кључне речи које се помињу на тим странама.
Према раније описаној схеми веб претраживања, потребно је да се направи листа (или мапа, што ће бити описано у каснијем току овог рада) **index**, која би се састојала од листи, које би опет имале једно поље за кључну реч и остала поља за хипервезе где се помиње та реч.
Дакле, прво се креира листа која садржи листу. Кључна реч ће бити у нултом пољу те листе, а у осталим ће постојати хипервезе везане за ту кључну реч (види слику 9).
После иницијалног дела могуће је даље додавати нове листе у индекс, где је почетно поље



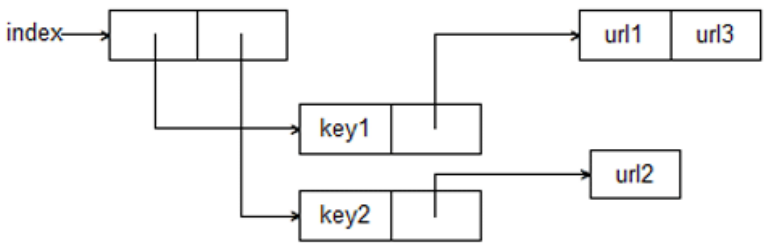
Слика 9: Креирање првог поља у индексу

такође кључна реч (слика 10).
Међутим, могуће је да постоји више хипервеза који су везани за исту кључну реч. У том



Слика 10: Следећа кључна реч и њена хипервеза

случају, додаје се ново поље листи у којој је та кључна реч (слика 11).



Слика 11: Два хиперлинка са истом кључном речју

За прављење и коришћење индекса, потребне су три процедуре:

add to index - процедура која додаје кључну реч и хипервезу у индекс;

lookup - враћа листу свих хипервеза на основу кључне речи;

add page to index - садржај странице (HTML кôд) раставља на кључне речи и убацује их у индекс.

У наредним поглављима биће размотрени кôдови поменутих процедура.

5.1.1 Прављење индекса

Процедура додавања кључних речи и хипервеза у индекс, узима три податка: index (листа) и две ниске - кључну реч (keyword) и хипервезу(url).

```
1 def add_to_index(index, keyword, url):
2     for entry in index:
3         if entry[0]==keyword:
4             entry[1].append(url)
5             return  # ne vraca se nista, samo se menja index
6     index.append([keyword, [url]])
```

Кôд 36: Процедура add_to_index

Ова процедура прелистава индекс и ако наиђе кључна реч која постоји у индексу, онда само додаје нову хипервезу у листу хипервеза, а ако кључна реч не постоји у индексу, додаје нову листу у индекс, са једном хипервезом.

5.1.2 Процедура за претраживање индекса

Процедура тражења узима два податка: index (листа) и кључну реч (ниска).

```
1 def lookup(index, keyword):
2     lookup_list = []
3     for entry in index:
4         if entry[0]==keyword:
5             for entry_url in entry[1]:
6                 lookup_list.append(entry_url)
7     return lookup_list
```

Кôд 37: Процедура lookup

Процедура прво прави привремену листу и претражује индекс, тражећи кључну реч. Када је нађе, онда сваку хипервезу везану за ту кључну реч, поставља у привремену листу и њу враћа као резултат.

5.1.3 Градња индекс листе

Сад кад је позната процедура add_to_index, могуће је дати кôд процедуре add_page_to_index, која узима три варијабле: индекс листу, хипервезу као ниску и садржај странице такође као ниску. Ова процедура служи за прављење индекса.

```
1  def add_page_to_index(index, url, content):
2      for entry in content.split():
3          add_to_index(index, entry, url)
```

Кôд 38: Процедура add_page_to_index

Процедура читав садржај странице раставља на речи (празан карактер је подразумевана вредност за сепаратор) и онда се сваку реч додаје у индекс заједно са хипервезом.

5.1.4 Промена веб-паука да ради са индексом

Пошто су наведене ове три кључне процедуре кôд измењеног веб-паука тако да ради са индексом.

```
1  def get_page(url):
2      try:
3          import urllib
4          return urllib.urlopen(url).read()
5      except:
6          return ""
7  def union(a, b):
8      for e in b:
9          if e not in a:
10             a.append(e)
11 def get_next_target(page):
12     start_link = page.find('<a href=')
13     if start_link == -1:
14         return None, 0
15     start_quote = page.find('"', start_link)
16     end_quote = page.find('"', start_quote + 1)
17     url = page[star_quote + 1: end_quote]
18     return url, end_quote
19 def get_all_pages(page):
20     while True:
21         links = []
22         url, endpos = get_next_target(page)
23         if url:
24             links.append(url)
25             page = page[endpos:]
26         else:
27             break
28     return links
29
30 # kreiranje indexa
```

```
31
32 def add_to_index(index, keyword, url):
33     for entry in index:
34         if entry[0] == keyword:
35             entry[1].append(url)
36             return
37     index.append([keyword, [url]])
38 def add_page_index(index, url, content):
39     for entry in content.split():
40         add_to_index(index, entry, url)
41
42 # glavna funkcija
43
44 def crawl_web(seed):
45     tocrawl = [seed]
46     crawled = []
47     index = [] # inicijalizacija indexa
48     while tocrawl:
49         page = tocrawl.pop()
50         if page not in crawled:
51             content = get_page(page)
52             add_page_to_index(index, page, content)
53             union(tocrawl, get_all_pages(content))
54             crawled.append(page)
55     return crawled
```

Кôд 39: Веб-паук који ради са индексом

Ако се овом кôду придода и кôд процедуре *lookup*, онда се добијају резултати у облику листе свих хипервеза које одговарају траженој кључној речи. Следи дата процедура која ради са индексом:

```
1 def lookup(index,keyword):
2     lookup_list = []
3     for entry in index:
4         if entry[0]==keyword:
5             for entry_url in entry[1]:
6                 lookup_list.append(entry_url)
7     return lookup_list
```

Кôд 40: Процедура lookup која ради са индексом

Већ сад је могуће добити све хипервезе у односу на тражену кључну реч. За резултате погледати поглавље 9.

5.2 Како убрзати?

У претходним поглављима овог рада је изграђен систем који може да одговори на упите, тако што ће проверити једну кључну реч из индекса у једном тренутку. Претраживач ће преко **lookup** процедуре проверити да ли у индексу постоји кључна реч која је постављена у упит и онда на основу тога дати одговарајући резултат.
Ипак, са великим индексом и већим бројем упита, овакав систем ће се испоставити као *спор*. Типични претраживач би требало да одговори на упите за мање од секунде, ако не и много брже. Закључак је да претраживач мора да ради брже са великим индексом.
Поставља се питање шта је потребно да би неки програм боље радио, одн. узимао што мање ресурса, а у исто време што брже достављао тражене резултате. Наравно, при томе не сме да се доведе у питање тачност и квалитет рада. Када програми постану велики, потребно је водити рачуна о томе колико "коштају" када их покренемо. Евалуација трошкова једног програма приликом његовог рада, је веома важна и представља један од највећих проблема у рачунарству. Поступак евалуације се назива *анализа алгоритма*[14]
Цена алгоритма зависи од улаза. Цена се касније испоставља у већој потрошњи ресурса рачунара. Претпоставимо да имамо два различита алгоритма Алго1 и Алго2, који успешно решавају исти проблем:

- Улаз $\longrightarrow$ Алго1 $\longrightarrow$ Резултат
- Улаз $\longrightarrow$ Алго2 $\longrightarrow$ Резултат

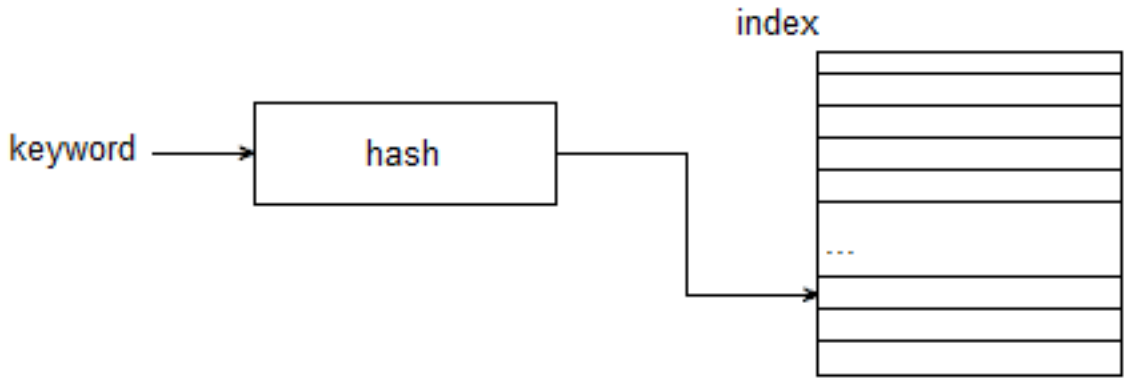
Није могуће поставити фиксну цену и рећи конкретну суму колико коштају сваки од ова два алгоритма. За неке улазе, Алго1 ће бити јефтинији него Алго2, али за друге улазе, ситуација ће бити другачија. Другим речима, потребно је предвидети цену, а да се не анализира алгоритам за сваки улаз.
Количина улазних података је главни фактор који утиче на брзину алгоритма, одн. цена извршења програма је директно пропорционална односа између повећања улазних података и повећања времена које је потребно да се изврши програм.
Дакле, потребно је знати која је цена (време, меморија) извршења једног програма у зависности од количине улазних података. Такође, потребно је посебно анализирати најгору могућу ситуацију приликом процеса претраживања. Са **lookup** процедуром и индексом чији кôд је наведен у претходном поглављу, најгори случај би био ако се кључна реч која се тражи у упиту налази на крају индекс листе. Тако да, независно од количине улазних података (индекс), не може се са сигурношћу предвидедти колико ће времена требати док се добије резултат, јер кључна реч може да се нађе одмах на почетку листе, где би се резултат добио готово одмах, али може и да се нађе на крају индекс листе, где је потребно претражити све елементе листе.

5.2.1 Хеш табела

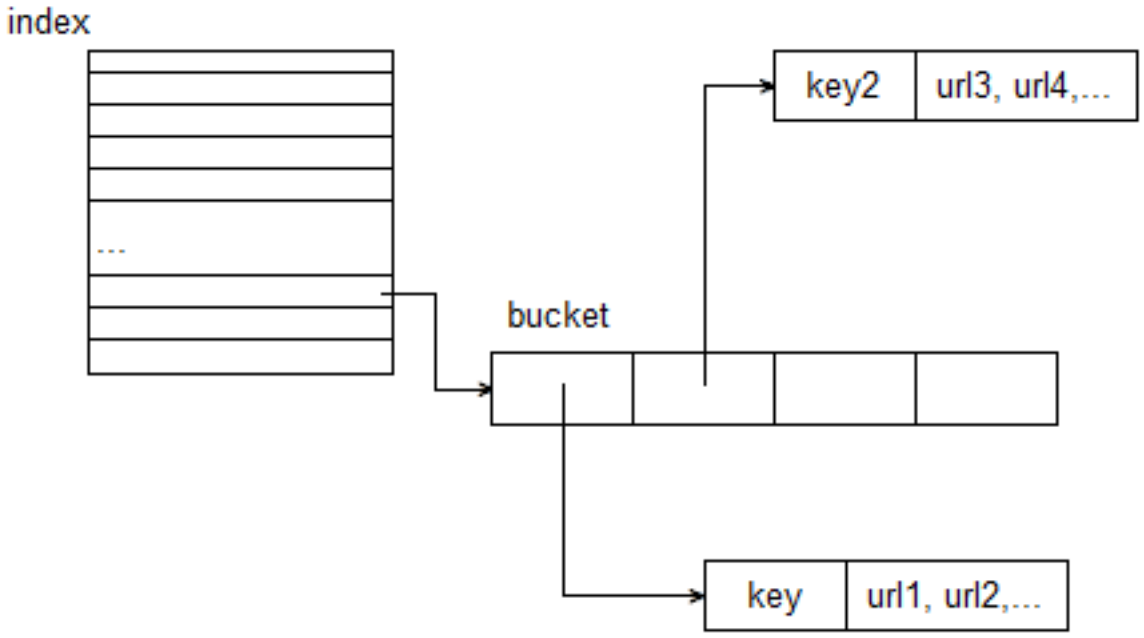
Уколико редослед елемената није битан, може се користити колекција која обезбеђује много бржи приступ елементима. Таква структура се назива хеш табела. Она организује елементе по сопственом редоследу. Наиме, сваки елемент добија свој број, хеш-кôд, који не зависи од осталих елемената. Битно је да се хеш-кôд брзо израчуна и да то израчунавање зависи само од елемента који се смешта у хеш табелу.

[14]Под алгоритмом подразумевамо било коју добро дефинисану процедуру која узима неку вредност или скуп вредности као *улаз* и као резултат даје неку вредност или скуп вредности као *излаз*[**?**]

Хеш табела се реализује као низ листи (или мапа). За проналажење и смештање елемента у хеш табелу израчунава се хеш-кôд. Резултат је индекс члана у низу, тј. индекс листе у коју ће бити смештен или која садржи дати елемент. Ако нема других елемената у тој листи, онда се елемент смешта на прво место у датој листи. Ако дође до тзв. *колизије*, тј. ако већ постоји елемент и/или више елемената у тој листи, тада се нови елемент пореди са осталима из дате листе и ако га већ нема у листи, додаје се на крај листе.



Слика 12: Хеш табела

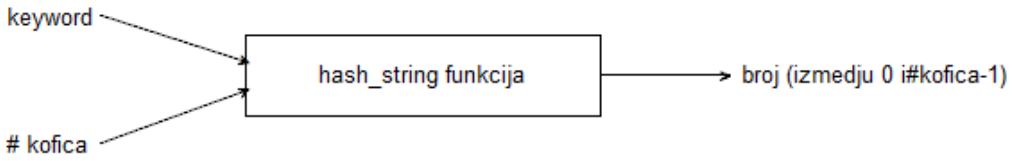


Слика 13: Смештање елемената у листу хеш табеле

Дакле, ако постоје k кључних речи и b листа у хеш табели, потребно је имати одговарајућу процедуру која ће смештати кључну реч у одговарајућу листу.

5.2.2 Дефинисање хеш функције

Претпоставимо да имамо b листи у хеш табели. Хеш функција треба да узме као улазне податке кључну реч и број листи, а да као излаз да вредност између 0 и b-1 (слика 14).



Слика 14: Hash функција

Најједноставније решење би било да функција узима прво слово кључне речи, налази *ASCII*[15] кодни број карактера који је прво слово кључне речи, затим дели тај број по модулу b (где је b број листи хеш табеле) и резултат тог дељења даје индексни број листе у коју ће кључна реч бити смештена.

```
1 def bad_hash_string (keyword, buckets):
2     return ord(keyword[0]%buckets)
```

Кôд 41: "Лоша" хеш функција

Да би се анализирало да ли је ово добар начин да се равномерно поделе кључне речи по листама у табели, потребно је тестирати ову функцију. Следећа функција ће послужити да би се видела расподела елемената по листама хеш табеле:

```
1 def test_hash_function(func, keys, size):
2     results = [0] * size
3     keys_used = []
4     for w in keys:
5         if w not in keys_used:
6             hv = func(w, size)
7             results[hv] += 1
8             keys_used.append(w)
9     return results
```

Кôд 42: Тестирање хеш функције

[15] скраћеница од „Амерички стандардни код за размену података“ (енгл. *American Standard Code for Information Interchange*) је кодни распоред заснован на латиничном писму које се користи у енглеском језику. Сваки карактер добија одговарајући број. На пример. "а" има број 98, "А" 67, "б" 99, итд.

Ова функција ће узети као улазне податке саму хеш функцију, кључне речи (као листу ниски) и број листи хеш табеле. Резултат који се добија није најбољи:

```
words = get_page('http://www.gutenberg.org/files/1497/1497.txt').split()
# uzimaju se sve reci iz Platonove "Republike"

counts = test_hash_function(bad_hash_string, words, 12)
# rezultat testiranja se smesta u promenljivu counts

print counts
[725, 1509, 1066, 1622, 1764, 834, 1457, 2065, 1398, 750, 1045, 935]
```

Кôд 43: Резултат тестирања лоше hash функције

Као што се види, у једној листи имамо 725 речи, а у другој преко 2000. То није добро, јер је циљ распоредити речи што уједначеније, а тиме и убрзати процес тражења кључних речи у хеш табели.
Бољи начин прављења хеш функције био би да се зависно од кључне речи, саберу ASCII кôдови свих знакова у тој речи, затим се тај збир подели по модулу броја листи хеш табеле и резултат дељења ће бити индексни број листе у коју се смешта кључна реч.

```
1 def hash_string(keyword, buckets):
2     h = 0
3     for c in keyword:
4         h = (h + ord(c))%buckets
5     return h
```

Кôд 44: Боља хеш функција

Ако се тестира нова хеш функција за исте улазне вредности, резултати ће бити нешто бољи:

```
counts = test_hash_function(hash_string, words, 12)
# rezultat testiranja nove funkcije
[1363, 1235, 1252, 1257, 1285, 1256, 1219, 1252, 1290, 1241, 1217, 1303]
```

Кôд 45: Тестирање "боље" хеш функције

Очигледно је расподела боља, мада свакако није идеална. Најмање попуњена листа има 1217 речи, а највише преко 1360. Тражење идеалне расподеле је предмет шире расправе, тако да ће за потребе овог рада, бити усвојена ова функција као пример добре расподеле.

5.2.3 Прављење празне хеш табеле

Ако је познат број листа хеш табеле, потребно је прво направити празну хеш табелу у коју ће се касније смештати подаци. Питање је какву структуру одабрати за ову табелу. Следи кôд који прави празну хеш табелу од n листи.

```
1  def make_hashtable(n):
2      i = 0
3      table = []
4      while i < n:
5          table.append([])
6          i = i + 1
7      return table
```

Кôд 46: Празна хеш табела

5.2.4 Налажење одговарајуће листе

Ако је потребно сместити податке у листе хеш табеле, онда мора бити познат индексни број листе у коју ће се сместити податак. Такође, кад се укаже потреба за претраживањем података, опет мора бити познат индексни број листе у којој се налази тражени елемент. Дакле, потребне су две процедуре, нпр. *add* и *lookup*, прва која ће да додаје елементе у хеш табелу, а друга која ће да тражи одговарајући податак у табели. Претпоставимо да већ постоје функције *hash_string*(видети кôд 44) и *make_hashtable*(видети кôд 46) које су раније наведене.

```
1  # pomocna funkcija: vraca indeksni broj liste u odnosu na keyword
2  def hashtable_get_bucket(htable, keyword):
3      return htable[hash_string(keyword, len(htable))]
4
5
6  # funkcija dodavanja u tabelu
7  def hashtable_add(htable, key, value):
8      return hashtable_get_bucket(htable, key).append([key, value])
9
10
11 # funkcija trazenja
12 def hashtable_lookup(htable, key):
13     bucket = hashtable_get_bucket(htable, key)
14     for i in bucket:
15         if i[0]==key:
16             return i[1]
17     return None
```

Кôд 47: *add* и *lookup* функције

На почетку наведеног кôда је дата помоћна функција *hashtable_get_bucket* јер се употребљава у обе тражене функције за исту сврху - враћање индексног броја листе за одговарајућу кључну реч.

5.2.5 Коришћење мапе уместо листе

Уместо листе за прављење хеш табеле, могуће је користити мапе (погледати под 2.3.3). Да би се имплементирала таква структура, потребно је преправити претходно наведене процедуре: **crawl_web**, **add_to_index** и **lookup**, тако да могу да раде са мапама, а не са листама. Предност мапа у односу на листе је прегледност и бржи рад. Тако да би алгоритам могао да буде реализован на следећи начин:

```
1  def get_page(url):
2      try:
3          import urllib
4          return urllib.urlopen(url).read()
5      except:
6          return ""
7
8  def union(p,q):
9      for e in q:
10         if e not in p:
11             p.append(e)
12
13 def get_next_target(page):
14     start_link = page.find('<a href=')
15     if start_link == -1:
16         return None, 0
17     start_quote = page.find('"', start_link)
18     end_quote = page.find('"', start_quote + 1)
19     url = page[start_quote + 1:end_quote]
20     return url, end_quote
21
22 def get_all_pages(page):
23     while True:
24         links = []
25         url, endpos = get_next_target(page)
26         if url:
27             links.append(url)
28             page = page[endpos:]
29         else:
30             break
31     return links
32
33 # index
34 def add_to_index(index,keyword,url):
35     if keyword in index:
36         index[keyword].append(url)
37     else:
38         index[keyword]=[url]
39
40 def add_page_to_index(index,url,content):
41     for entry in content.split():
```

```
42          add_to_index(index, entry, url)
43
44  # lookup
45  def lookup(index, keyword):
46      if keyword in index:
47          return index[keyword]
48      else:
49          return None
50
51
52  def crawl_web(seed):
53      tocrawl = [seed]
54      crawled = []
55      index = {} #inicijalizacija mape
56      while tocrawl:
57          page = tocrawl.pop()
58          if page not in crawled:
59              content = get_page(page)
60              add_page_to_index(index, page, content)
61              union(tocrawl, get_all_pages(content))
62              crawled.append(page)
63      return crawled
```

Kôд 48: Мапа уместо листе

6 Рангирање страница

Пошто је завршен кôд *crawl_ web* процедуре и када постоји веб-паук који може на задовољавајући начин да скенира странице, смешта линкове у индекс или хеш табелу и да даје резултате у односу на постављене упите, следећи корак је рангирање страница. Рангирање страница је и најзахтевнији део веб претраживача. На почетку поглавља је поменут алгоритам за рангирање страница $PageRank^tm$, који је срце Гугловог претраживача. У оквиру овог рада реализоваће се алгоритам сличан $PageRank^tm$ алгоритму у Python програмском језику.

6.1 Такмичење у популарности

Поставља се питање: ко је популаран? Шта је популарност? Ако Ана има највише пријатеља да ли је она најпопуларнија? У школи, на пример, није најважније имати пуно пријатеља, поготово ако су ти пријатељи особе које немају много пријатеља. Важно је и да пријатељи буду популарни. Такође, треба имати у виду и да особа која има јако пуно пријатеља то пријатељство баш и не цени, те тиме и популарност опада.
Теза оснивача Гугла, Брина и Пејџа, каже[**?**, Ch 4]

Страница је важна ако се на њу показује са других важних страница.

Прва верзија формуле за израчунавање популарности у PageRank алгоритму је сличила следећем:

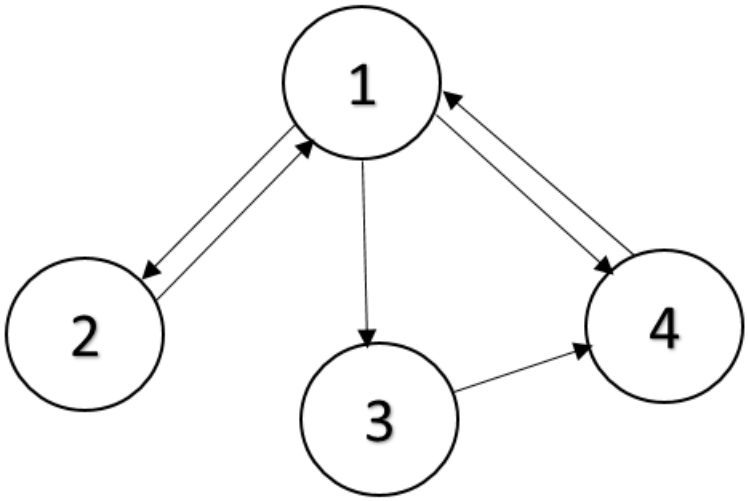
$$rank(P_i) = \sum_{P_j \in B_{P_i}} \frac{rank(P_j)}{|P_j|} \tag{1}$$

где је P_i i-та страница, B_{P_j} скуп страна које показују на P_j, $|P_j|$ број излазних линкова од P_j. Проблем са формулом 1 је у томе што не може да се одреди број P_j, одн. број линкова од P_j ка P_i. Проблем је решен коришћењем итеративне методе. Дакле, у почетку све странице имају ранг $\frac{1}{n}$, где је n, број страница у Гугловом индексу. Тако да се сада рачуна сваки $rank(P_i)$ за сваку страницу P_i из индекса и онда само треба одредити после колико итерација k ће $rank_{k+1}(P_i)$ бити довољно прецизан. Тако да формула гласи[**?**, Ch 4.1]

$$rank_{k+1}(P_i) = \sum_{P_j \in B_{Pj}} \frac{rank_k(P_j)}{|P_j|} \tag{2}$$

где је $rank_0(P_i) = \frac{1}{n}$ за сваку страницу из индекса.
На пример, ако се претпостави да постоји овакав оријентисан граф на слици 15, који представља модел четири веб странице, где су хипервезе представљане уређеним везама.

Израчунавање ранга у две итерације је представљано у табели 5.



Слика 15: Оријентисани граф модела четири веб странице

итерација 0	**итерација 1**	**итерација 1**	**PageRank**
$rank(1) = \frac{1}{4}$	$\frac{1}{2}$	$\frac{5}{12}$	I
$rank(2) = \frac{1}{4}$	$\frac{1}{12}$	$\frac{1}{6}$	III
$rank(3) = \frac{1}{4}$	$\frac{1}{12}$	$\frac{1}{6}$	III
$rank(4) = \frac{1}{4}$	$\frac{1}{3}$	$\frac{1}{4}$	II

Табела 5: Израчунавање PageRank-а у две итерације

6.2 Матрични облик

Аутори PageRank-а су заменили суму вектором и од вектора направили матрични модел. Нека је H матрица димензија $m \times n$, таква да је $H_{ij} = \frac{1}{|P_i|}$, ако постоји линк од i ка j или 0 у супротном. Тада би матрица графа била:

$$H = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

Нека је π^T вектор, такав $\pi^{(k)T}$ представља PageRank вектор у итерацији k. Тада се следећа итерација добија као:

$$\pi^{(k+1)T} = \pi^{(k)T} H \tag{3}$$

Свака итерација захтева једно множење вектора и матрице, што нам даје $O(n^2)$ израчунавања. Но, како је матрица H са врло мало не-нула поља, она захтева $O(nnz(H))$ израчунавања, где

је $nnz(H)$ број ненула елемената матрице H. Такође, матрица H веома личи на транзициону стохастичку матрицу Марковљевих ланаца[16] тако да је можемо назвати субстохастичком[**?**, Ch 4.2]. Брин и Пејџ нису користили термин "Марковљев ланац". Али оно што јесу урадили је да су матрицу H врло мало модификовали како би она била стохастичка. Уместо термина "Марковљев ланац", користили су термин "случајни сурфер"(енгл. *random surfer*). Када такав сурфер дође на страницу са неколико хипервеза, он бира једну од њих на случајан начин и наставља тај процес унедоглед. У дужем временском периоду, део времена који сурфер проводи на датој веб страници је мерило о релативној важности те странице. Ако он проводи доста времена на некој страници, онда мора да се стално враћа на ту страну. Странице које он поново посећује морају бити важне зато што на њих показују друге важне странице.
Два су главна проблема са иницијалним *случајним сурфером*:

dangling node овај термин означава чвор графа који не показује ни на један други чвор

конвергенција да ли ће и после колико итерације, алгоритам дати исправне и очекиване резултате

Као резултат решавања првог проблема, добија се модификована матрица која решава тај проблем, уз помоћ :

$$S = H + a(\frac{1}{n}e^T) \tag{4}$$

где је e^T јединични вектор, а $a = \begin{cases} 1, & dangling\ node \\ 0, & inace \end{cases}$ и n остаје број страница у индексу.
Други проблем, тј. проблем конвергенције, је решен тиме што се матрица S модификовала тако да она постане стохастичка, несводљива, апериодична и примитивна, те тако и конвергира. Модификација матрице S ради се у неколико корака:

$$G = \alpha S + (1-\alpha)\frac{1}{n}ee^T \tag{5}$$

$$G = \alpha(H + a(\frac{1}{n}e^T) + (1-\alpha)\frac{1}{n}ee^T \tag{6}$$

$$G = \alpha H + (\alpha a + (1-\alpha)e)\frac{1}{n}e^T \tag{7}$$

где је $0 \leqslant \alpha \leqslant 1$ параметар који даје својство *телепортације* случајном сурферу, тј могућност да почне процес из почетка. На пример ако је $\alpha = 0.6$ то значи да ће 60% времена случајни сурфер проводити време кликћући линкове, а 40% времена ће почињати процес из почетка, бирајући случајним одабиром нову страницу за почетак.
У примеру са слике 15 после краћег рачунања добија се следећа матрица G:

$$G = \begin{bmatrix} 0.025 & 0.325 & 0.325 & 0.325 \\ 0.925 & 0.025 & 0.025 & 0.025 \\ 0.025 & 0.025 & 0.025 & 0.925 \\ 0.925 & 0.025 & 0.025 & 0.025 \end{bmatrix}$$

[16]Марковљеви ланци описују стохастичке системе(*стохастичке или вероватносне*, прим. аут.) "без меморије", тј. такве системе код којих вероватноће будућих стања зависе само од садашњег, а не од прошлих стања.[**?**]

6.3 Израчунавање вредности PageRank™ вектора

Вектор π^T је могуће добити на два начина, уз услов да је $\pi^T e = 1$:

1. Налажењем леве сопствене вредности матрице G, тј $\pi^T G = \lambda \pi^T$
2. Налажењем левог нула вектора од $I - G$, тј. $\pi^T (I - G) = O^T$

У примеру са слике 15 добија се: $\pi^T = [0.7608\ 0.2740\ 0.2740\ 0.5206]$[17]

```
1  import numpy as np
2  from scipy.linalg import eig
3
4  T = np.mat("0.025  0.325  0.325  0.325;
5              0.925  0.025  0.025  0.025;
6              0.025  0.025  0.025  0.925;
7              0.925  0.025  0.025  0.025")
8
9  values, left = eig(T, left = True, right = False)
10
11 for i in range(len(values)):
12         print("Levi sopstveni vektor za sopstvenu vrednost {}:".format(values[i]))
13         print(left[:,i])
14         print()
15 >>>Levi sopstveni vektor za sopstvenu vrednost (0.9999999999999998+0j):
16 [-0.76083506+0.j  -0.27398492+0.j  -0.27398492+0.j  -0.52057136+0.j]
```

Кôд 49: Израчунавање леве сопствене вредности

Тумачећи резултате, долази се до закључка да је страница **1** најпопуларнија, јер има највећу вредност, друга је страница **4**, а треће место деле странице **2** и **3**.

[17] Израчунато у Python-у, помоћу модула numpy - `www.numpy.org` и scypy `www.scypy.org`

6.4 Рангирање веб страница у Python-у

Имплементација ове варијанте PageRank алгоритма у crawl_web модул се врши додавањем још једне структуре - граф, како би се знало на који начин је случајни сурфер прегледавао странице. Граф ће се једноставно иницијализовати у самој функцији crawl_web, као празна мапа. Граф би требао садржи хипервезу као кључ и листу хипервеза на које показује, као вредност. Тако да измењена функција crawl_web изгледа овако:

```
1  def crawl_web(seed):
2      tocrawl = [seed]
3      crawled = []
4      graph = {}  # <url>, [lista stranica na koje pokazuje]
5      index = {}
6      while tocrawl:
7          page = tocrawl.pop()
8          if page not in crawled:
9              content = get_page(page)
10             add_page_to_index(index, page, content)
11
12             outlinks = get_all_links(content)
13             graph[page]=outlinks # pravljenje grafa
14
15             union(tocrawl, outlinks)
16             crawled.append(page)
17     return index, graph
```

Кôд 50: Увођење графа у crawl_web

6.5 Израчунавање ранга странице

Ранг странице ће у овом раду бити израчунат помоћу формуле 2, измењену за *dumping* константу, која је позитивна и није већа од 1. Познато је и да је почетна вредност свих страница у нултом кораку итерације иста и износи $\frac{1}{n}$. Дакле,

$$rank(0, P_i) = \frac{1}{n} \tag{8}$$

$$rank(k+1, P_i) = \alpha \sum_{P_j \in B_{P_j}} \frac{rank(k, P_j)}{|P_j|} + (1-\alpha)\frac{1}{n} \tag{9}$$

Оваква поставка намеће рекурзију као решење, али како није унапред познат број рачунања, тиме је опрезније применити итерацију приликом рачунања. Уз листинг кода 48 и уз измењену функцију crawl_web из листинга 50, сада ће се придодати и функција за израчунавање ранга странице *rank_ compute*, која узима граф као улаз и враћа мапу, чија је кључна реч хипервеза, а вредност је њен ранг.

```
def compute_ranks(graph):
    alfa = 0.9 # damping faktor
    numloops = 10 # broj iteracija

    ranks = {}
    npages = len(graph)
    for page in graph:
        ranks[page] = 1.0 / npages

    for i in range(0, numloops):
        newranks = {}
        for page in graph:
            newrank = (1 - alfa) / npages

            for node in graph:
                if page in graph[node]:
                    newrank += ranks[node]*alfa/len(graph[node])

            newranks[page] = newrank
        ranks = newranks
    return ranks
```

Кôд 51: Израчунавање ранга странице

На крају остаје још да се сортирају резултати и да се омогући кориснику да страницу са највећим рангом види као прву. Ради тога се уводи функција *results*, која ће узети индекс, рангове који су постављени у мапу и наравно, кључну реч корисника. Као резултат се даје листа са свим релевантним хипервезама који су поређани по рангу.

```
def results(index, ranks, keyword):
    urls = lookup(index, keyword)
    if urls == None:
        return None
    results_url = []
    results_num = []
    for e in urls:
        results_num.append(ranks[e])
    results_num.sort()
    while results_num:
        current_max = results_num.pop()
        for url in urls:
            if ranks[url]==current_max:
                results_url.append(url)
                break
    return results_url
```

Кôд 52: Функција која враћа најбољи резултат

Овим је у потпуности реализован алгоритам за претраживање веба. Кôд је у целости дат у Додатку А (видети поглавље 8).

7 Закључак

Програмски језик Python поседује разноврстан избор типова података, структура, омогућава програмеру да у кратком временском периоду направи моћан програм. Python поседује све што модеран програмски језик мора да има. Но, он има још више. Армију корисника који учествују у даљем развоју језика својим саветима и расправама на интернет форумима и групама. Python је веома "жив" програмски језик, који периодично избацује нове верзије и побољшава перформансе.

Додаци које Python доноси са својим модулима и проширењима чине га конкурентним у односу на комерцијалне програмске језике какви су Java, C#...Може се користити и као скрипт језик у оквиру прављења веб апликација, може се користити за писање мањих делова програма у C/C++, а може се користити и самостално за писање десктоп и веб апликација. Данас је незаменљиво оруђе у рукама научника, који га користе за компликоване прорачуне, анализу и презентацију података.

Проширење у виду *django framework*-а[18] омогућава широку употребу Python-а у писању веб апликација, пре свега CMS[19], али и мањих веб страница.

Програмски језик Python у овом раду је у потпуности одговорио на потребе писања веб претраживача. Током писања кôда, коришћене су само основне функције Python-а. Нису се употребљавале компликоване и робустне структуре, што омогућава потенцијалном читаоцу лак увид у кôд и процену шта ће кôд на крају даје као резултат.

Процес који је описан на почетку поглавља 3.3 успешно је окончан. Оно што превазилази тему овог рада је даља имплементација веб претраживача у веб апликацији. Међутим, може се закључити да је претраживање са укљученим модулом рангирања далеко подесније, него ли оно које само испоставља списак линкова без претходног рангирања (в. поглавље 9).

Веб претраживање данас је уносан посао. Компаније које нуде услуге веб претраживања наплаћују другим фирмама за резултате својих моћних веб-паукова, као и за трошкове рекламирања, на пример. У модерном добу, где је компанија Гугл оставила штампане енциклопедије на ропотарницу историје, неопходно је имати тачан и поуздан систем веб претраживања. Такође, резултати претраживања сугеришу компанијама на који начин ће лакше доћи до купца. У овом тренутку расте потреба за таквим видом информација. Процес таквог прикупљања информација, назива се *data mining*(енгл., *ископавање података*), где се у процесу сакупљања огромне количине података са Интернета, покушавају донети закључци у циљу бољег функционисања компаније.

На самом крају може се закључити да се предлаже примена језика Python за реализацију алгоритама којим се врши рангирање веб страница. На крају је са успехом имплементиран Python програм за претраживање веба и рангирање страница.

[18] погледати `http://www.djangoproject.com/`

[19] Content Management System

8 Додатак А: Завршни кôд

Завршни кôд веб претраживача написан у Python-у:

```
'''
@author: Igor Ilic
'''
max_pages = 100

'''
modul veb-pauka
uzima url
vraca index i graph, kao mape
'''
def crawl_web(seed): # vraca index, graph inlinks
    tocrawl = [seed]
    crawled = []
    graph = {}  # hiperveza, [lista stranica na koje pokazuje]
    index = {}
    count = 0
    while tocrawl and count<max_pages:
        page = tocrawl.pop()
        if page not in crawled:
            content = get_page(page)
            add_page_to_index(index, page, content)
            outlinks = get_all_links(content) # linkovi ka
            graph[page]=outlinks #pravi se graph
            union(tocrawl, outlinks)
            crawled.append(page)
            count+=1
    return index, graph
# get_page uzima url i vraca html kod stranice
def get_page(url):
    try:
        import urllib
        return urllib.urlopen(url).read()
    except:
        return ""
# get_next_target uzima kod stranice
# vraca prvu hipervezu na koju nailazi
# i poziciju u kodu stranice na kojoj prestaje url
def get_next_target(page):
    start_link = page.find('<a href=')
    if start_link == -1:
        return None, 0
    start_quote = page.find('"', start_link)
    end_quote = page.find('"', start_quote + 1)
    url = page[start_quote + 1:end_quote]
```

```
45      return url, end_quote
46  # get_all_links uzima kod stranice
47  # vraca sve hiperveze u obliku liste
48  def get_all_links(page):
49      links = []
50      while True:
51          url, endpos = get_next_target(page)
52          if url:
53              links.append(url)
54              page = page[endpos:]
55          else:
56              break
57      return links
58  # pomocna funkcija, sluzi za pravljenje unije dve liste
59  def union(a, b):
60      for e in b:
61          if e not in a:
62              a.append(e)
63  # add_page_to_index uzima index(mapa), hipervezu
64  # i listu kljucnih reci iz koda stranice
65  # dodaje sve stranice i kljucne reci u index
66  def add_page_to_index(index, url, content):
67      words = content.split()
68      for word in words:
69          add_to_index(index, word, url)
70  # add_to_index dodaje par kljucna rec, hiperveza
71  # u indeks
72  def add_to_index(index, keyword, url):
73      if keyword in index:
74          index[keyword].append(url)
75      else:
76          index[keyword] = [url]
77  # lookup trazi kljucnu rec u indexu
78  # vraca hiperveze koje odgovaraju kljucnoj reci
79  def lookup(index, keyword):
80      if keyword in index:
81          return index[keyword]
82      else:
83          return None
84  '''
85  modul rangiranja
86  uzima graf
87  vraca mapu sa parovima: hiperveze i ranga
88  '''
89  def compute_ranks(graph):
90      d = 0.8 # damping factor
91      numloops = 10
92
```

```
93      ranks = {}
94      npages = len(graph)
95      for page in graph:
96          ranks[page] = 1.0 / npages
97
98      for unused in range(0, numloops):
99          newranks = {}
100         for page in graph:
101             newrank = (1 - d) / npages
102
103             for node in graph:
104                 if page in graph[node]:
105                     newrank += ranks[node]*d/len(graph[node])
106
107             newranks[page] = newrank
108         ranks = newranks
109     return ranks
110 # sortira hiperveze po rangu koji imaju pocev od najpopularnije
111 def rank_list(ranks):
112     return sorted(ranks, key=ranks.__getitem__, reverse=True)
113 '''
114 modul rezultata
115 uzima index, rang i kljucnu rec
116 vraca hipervezu koja odgovara kljucnoj reci i ima najveci rang
117 '''
118 def lucky_search(index, ranks, keyword):
119     url_list = lookup(index, keyword)
120     if url_list == None:
121         return None
122
123     key = ''
124     maximum = 0
125     for entry in url_list:
126             if ranks[entry]>= maximum:
127                 maximum = ranks[entry]
128                 key=entry
129     return key
130
131 '''
132 primer
133 '''
134 index, graph = crawl_web('http://localhost/prva.html')
135 print(rank_list(compute_ranks(graph)))
```

Kôд 53: Завршни кôд

9 Додатак Б: Резултати

Да би се добила слика о значају рангирања страница, направљен је модел четири повезане веб странице, чији је граф дат на слици 15. Кôдови страница изгледају овако:

```
<html>
<head>
 <title>Prva stranica</title>
 <link rel="stylesheet" href="my.css">
</head>
<body>
 <h1>Prva stranica</h1>
 <ul>
  <li><a href="http://localhost/druga.html">Druga stranica</a></li>
  <li><a href="http://localhost/treca.html">Treca stranica</a></li>
  <li><a href="http://localhost/cetvrta.html">Cetvrta stranica</a></li>
 </ul>
 <p>Prva stranica ima hipervezu ka drugoj stranici, trecoj stranici,
 cetvrtoj stranici. Ova stranica se realizuje u cilju testiranja veb
 pretrazivaca.</p>
</body>
</html>
```

Кôд 54: Кôд прве странице

```
<html>
<head>
 <title>Druga stranica</title>
 <link rel="stylesheet" href="my.css">
</head>
<body>
 <h1>Prva stranica</h1>
 <ul>
  <li><a href="http://localhost/prva.html">Prva stranica</a></li>
 </ul>
 <p>Druga stranica ima hipervezu ka prvoj stranici. Ova stranica se realizuje u cil
 pretrazivaca.</p>
</body>
</html>
```

Кôд 55: Кôд друге странице

```
<html>
<head>
 <title>Treca stranica</title>
 <link rel="stylesheet" href="my.css">
</head>
<body>
 <h1>Treca stranica</h1>
 <ul>
  <li><a href="http://localhost/prva.html">Cetvrta stranica</a></li>
 </ul>
 <p>Treca stranica ima hipervezu ka cetvrtoj stranici. Ova stranica se realizuje u
 pretrazivaca.</p>
</body>
</html>
```

Kôд 56: Kôд треће странице

```
<html>
<head>
 <title>Cetvrta stranica</title>
 <link rel="stylesheet" href="my.css">
</head>
<body>
 <h1>Cetvrta stranica</h1>
 <ul>
  <li><a href="http://localhost/prva.html">Prva stranica</a></li>
 </ul>
 <p>Cetvrta stranica ima hipervezu ka prvoj stranici. Ova stranica se realizuje u c
 pretrazivaca.</p>
</body>
</html>
```

Kôд 57: Kôд четврте странице

Странице су затим постављене на локални сервер. Кад се уз кôд 53 пусти наредба: **print**(lookup(index, 'stranica'), добија се следећи резултат:

```
['http://localhost/prva.html',
 'http://localhost/prva.html',
 'http://localhost/cetvrta.html',
 'http://localhost/cetvrta.html',
 'http://localhost/treca.html',
 'http://localhost/treca.html',
 'http://localhost/druga.html',
 'http://localhost/druga.html']
```

Очигледно је да обично прегледање без рангирања даје списак свих хипервеза који одговарају датој кључној речи. Сада ће уз исти кôд бити дата наредба:

print(compute_ranks(graph)) и тада се добија

```
{'http://localhost/treca.html': 0.1572993996378601,
 'http://localhost/cetvrta.html': 0.28313891934814817,
 'http://localhost/prva.html': 0.40226228137613174,
 'http://localhost/druga.html': 0.1572993996378601}
```

У овом случају, испоставља се мапа у којој су кључеви хипервезе, а вредности њихов ранг у моделу.
Ако је потребно поређати хипервезе по редоследу рангирања, онда се наредбом:
print(rank_list(compute_graph(graph))), добија листа

```
['http://localhost/prva.html',
 'http://localhost/cetvrta.html',
 'http://localhost/treca.html',
 'http://localhost/druga.html']
```

И за сам крај, могуће је доставити кориснику тачно једну хипервезу, која одговара упиту и која има највећи ранг од свих хипервеза које одговарају упиту.
print(lucky_search(index, compute_ranks(graph), 'stranica')), тако да је резултат само прва страница.

```
http://localhost/prva.html
```

Може се извући закључак да је далеко корисније претраживање са укљученим модулом рангирања, које омогућава кориснику да за свој задати упит добије као резултат једну или више рангираних страница и тиме добије адекватну информацију.

10 Додатак B: Списак програмских кôдова употребљених у раду

Следи списак кôдова који су коришћени у овом раду.

Програмски кôдови