SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Projekt iz predmeta Bioinformatika

PORAVNAVANJE KRATKIH NIZOVA S DUGAČKIM KORISTEĆI DFA (AHO-CORASICK ALGORITAM)

Janja Paliska, Antonio Soldo, Antonio Zemunik

Voditelj: *doc. dr. sc. Mile Šikić*

Zagreb, siječanj, 2014.

SADRŽAJ

1. UVOD

Aho-Corasick algoritam omogućava efikasno pronalaženje konačnog broja ključnih riječi u dugačkim sekvencama teksta koristeći deterministički konačan automat (engl. *Deterministic finite automaton,* skraćeno DFA). Jednom izgrađeni automat nad konačnim skupom ključnih riječi može se dalje koristiti za obradu bilo kojeg teksta u samo jednom prolazu kroz tekst. [1]

Zadatak ovog projekta je implementacija navedenog algoritma u tri različita programska jezika (C#, Java, Python) te usporedba dobivenih rezultata. U nastavku je detaljnije objašnjen sam algoritam i implementacija te su dane upute za instalaciju za pojedini jezik.

2. ALGORITAM

Cijeli algoritam sastoji se od toga da se prvo izgradi DFA koristeći konačan skup ključnih riječi, a zatim se izgrađenom automatu kao ulaz predaje tekst u kojem se te ključne riječi traže. Izlaz iz automata su indeksi teksta na kojima su ključne riječi pronađene pri čemu se one u tekstu mogu i preklapati.

2.1. IZGRADNJA AUTOMATA

Konačan skup ključnih riječi nad kojima se automat izgrađuje označava se kao

$$K = \{y_1, y_2, ..., y_k\}$$

gdje je y_i pojedina ključna riječ.

Ulazni tekst u kojem se riječi pretražuju označava se kao x pri čemu je x niz znakova (engl. *string*). Skup znakova koji se pojavljajuju u tekstu i ključnim riječima označen je s $\sum$.

DFA se sastoji od skupa stanja određenih brojevima iz skupa N_0 pri čemu je početno stanje označeno brojem 0. Prijelazi iz jednog stanja u drugo određeni su dvjema funkcijama: *goto* funkcijom g i *failure* funkcijom f dok je ispis određen *output* funkcijom.

Goto funkcija definira parove oblika (stanje s, simbol a) i pridjeljuje im novo stanje s' u koje automat prelazi (ukoliko prijelaz za određeni par nije definiran, automat mu umjesto prijelaznog stanja dodjeljuje oznaku *fail*): $g(s, a) = s' / fail$

Za početno stanje definiramo prijelaz $g(0, a_i) = 0$ za svaki $a_i \in \sum$ za koji ne postoji drukčiji prijelaz iz 0 ranije definiran.

Failure funkcija svakom stanju s određuje stanje s_f , $f(s) = s_f$, u koje se prelazi za svaki simbol $a_i \in \sum$ za koje $g(s, a_i)$ nije definiran, odnosno za koje $g(s, a_i) = fail$. Jedino stanje za koje f nije definiran je početno stanje.

Output funkcija služi za ispis rezultata te je definirana u onim stanjima u kojima se pronalazi bilo koja ključna riječ $y_i \in K$: $output(s) = K_i$ pri čemu je K_i skup ključnih riječi koje završavaju u stanju s.

2.2. IZGRADNJA GOTO FUNKCIJE

Izgradnja *goto* funkcije odgovara izgradnji usmjerenog grafa nad ključnim riječima iz K pri čemu čvorovi grafa odgovaraju stanjima automata, a grane simbolima iz abecede $\sum$. Za početak se definira početno stanje, odnosno čvor 0. Dalje se za svaku ključnu riječ $y_i = a_1 \ldots a_n$ kreće iz stanja 0 te se počevši od prvog znaka a_1, dodaje nova grana obilježena znakom a_1 koja vodi od čvora 0 prema idućem čvoru. Postupak ponavljamo redom za sve preostale znakove iz dobivenog stanja nadalje. Kad god je moguće, izbjegava se stvaranje novih grana i čvorova (prvo se traži najveći prefiks riječi otprije dodan u graf). Svaki put, kada se dodavanje neke riječi završi, zadnjem čvoru (stanju) koji je stvoren dodaje se *output* funkcija s vrijednošću te ključne riječi.

- **Ulaz:** $K = \{y_1, y_2, \ldots, y_k\}$
- **Izlaz:** g funkcija, djelomična funkcija *output*
- **Pseudokod:**

```
novoStanje = 0
i = 1 : k
    g = enter (y_i)
za svaki a takav da g(0, a) = fail
    g(0, a) = 0
```

```
funkcija enter(a1a2…am)
  s = 0
  j = 1
  pronađi najduži prefiks od a1a2…am koji već postoji u g:
  dok g(s, aj) ≠ fail
      s = g(s, aj)
      j = j + 1
  za ostatak aj…am stvori nove prijelaze
  p = j : m
      novoStanje = novoStanje + 1
      g(s, ap) = novoStanje
      s = novoStanje
  output(s) = a1a2…am
```

2.3. IZGRADNJA FAILURE FUNKCIJE

Failure funkcija gradi se pomoću *goto* funkcije. Prvo se definira $f(s) = 0$ za stanja dubine jedan (dubina(s) je broj prijelaza iz 0 do s), a onda se na temelju toga izračunava f za stanja dubine dva itd., odnosno za stanja dubine d, *failure* funkcija računa se pomoću stanja dubine d-1.

- **Ulaz:** g, *output*
- **Izlaz:** f, *output*
- **Pseudokod:**

```
red = []
za svaki a takav da g(0, a) = s ≠ 0
    dodaj na red stanje s
    f(s) = 0
dok je red ≠ []
    r = sljedeće stanje iz reda
    makni iz reda r
    za svaki a takav da g(r, a) = s ≠ fail
        dodaj u red stanje s
        s' = f(r)
        dok je g(s', a) = fail
            s' = f(s')
        f(s) = g(s', a)
        output(s) = output(s) U output(f(s))
```

2.4. RAD AUTOMATA

- **Ulaz:** ulazni niz $x = a_1 \ldots a_n$ (a_i i-ti simbol niza), DFA s izgrađenim g, f i *output* funkcijama
- **Izlaz:** indeksi teksta x na kojima su ključne riječi pronađene
- **Pseudokod:**

```
s = 0
i = 1 : n
    dok je g(s, a_i) = fail
        s = f(s)
    s = g(s, a_i)
    ako postoji output(s)
        za svaku riječ y_i u output(s) ispiši:
            i – duljina(y_i), y_i
```

2.5. RAD AUTOMATA NA PRIMJERU

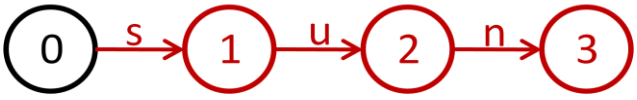
U ovome poglavlju prikazan je rad automata na konkretnom primjeru.

K = {sun, ms, sm, sms} → skup ključnih riječi

1) Stvaramo čvor za početno stanje 0

Slika 1. Stvaranje početnog stanja

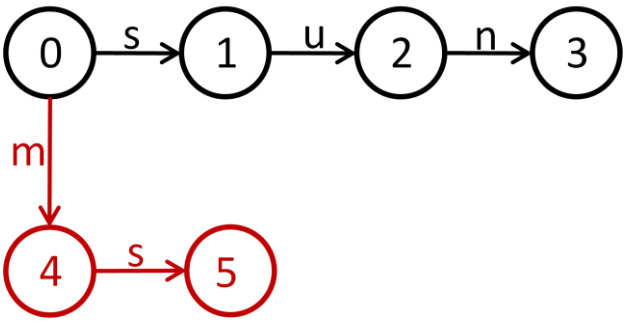
2) Dodaj potrebne prijelaze za K_1 = sun



Slika 2. Dodan prijelaz za "sun"

output(3) = {sun}

3) Dodaj potrebne prijelaze za K_2 = ms

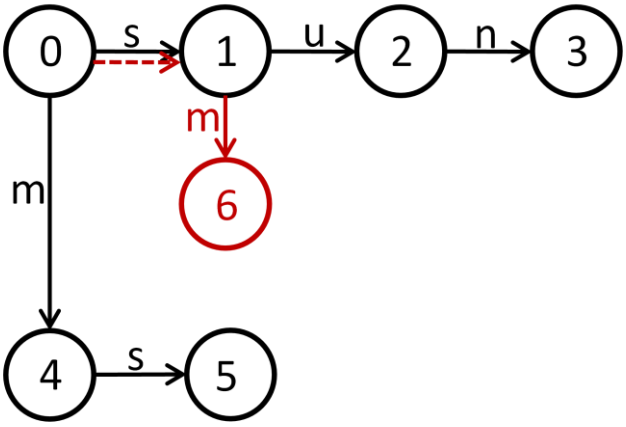


Slika 3. Dodan prijelaz za "ms"

output(3) = {sun}

output(5) = {ms}

4) Dodaj potrebne prijelaze za K_3 = sm



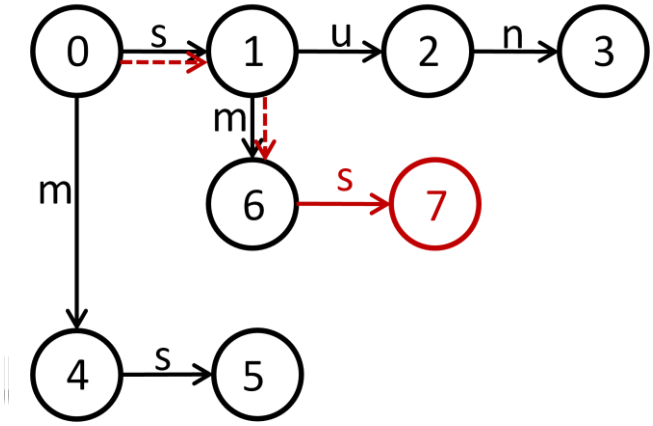
Slika 4. Dodan prijelaz za "sm"

output(3) = {sun}

output(5) = {ms}

output(6) = {sm}

5) Dodaj potrebne prijelaze za K_4 = sms



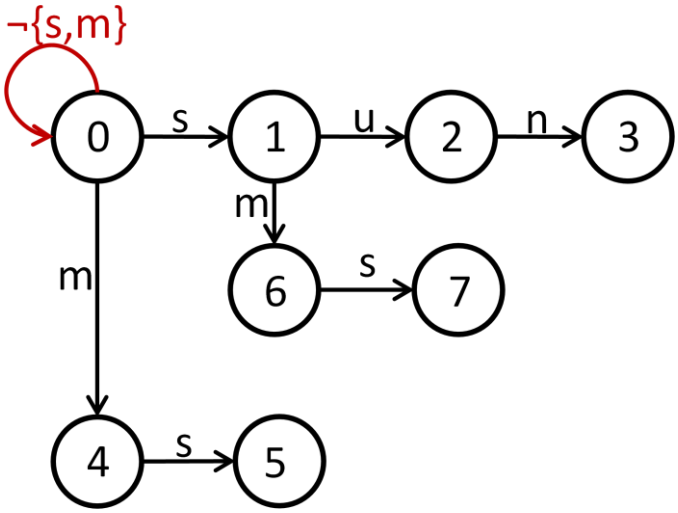
Slika 5. Dodani prijelazi za "sms"

output(3) = {sun}

output(5) = {ms}

output(6) = {sm}

output(7) = {sms}



Slika 6. Konačan izgled goto funkcije

6) Izračun *failure* funkcije

Tablica 1. *Failure* funkcija stanja automata

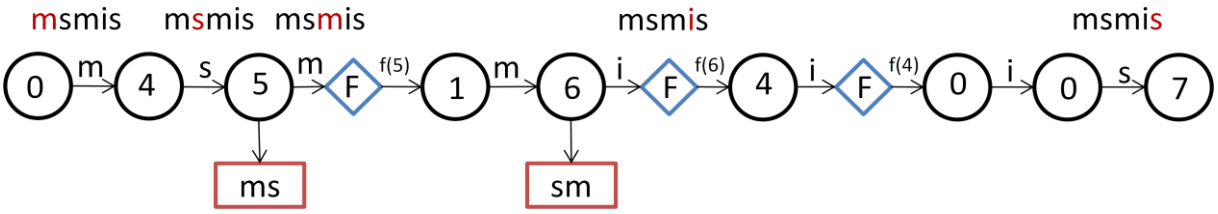
s	1	2	3	4	5	6	7
f(s)	0	0	0	0	1	4	5

7) Nadopuni *output* funkciju prijelazima iz *f*

Tablica 2. *Output* funkcija stanja automata

s	output(s)
3	sun
5	ms
6	sm
7	sms, ms

x = msmis



Slika 7. Prolazak x kroz automat

3. IMPLEMENTACIJA

Algoritam je implementiran u tri programska jezika: C#, Java, Python, a prilagođen je za rad na Bio-Linux platformi (Ubuntu 12.04).

Svaka od implementacija kao ulaz prima dva parametra: imena datoteke s ključnim riječima i datoteke s tekstom. Obje datoteke su tekstualnog tipa, a ključne riječi su u datoteci odvojene jednim zarezom (bez razmaka nakon zareza).

Nakon završetka, program ispisuje vrijeme potrebno za konstrukciju automata, vrijeme potrebno za pronalazak ključnih riječi (u to vrijeme uključeno je i učitavanje datoteke s tekstom) i ukupnu utrošenu memoriju. Uz to, program stvara izlaznu datoteku 'output' koja sadrži sve pronađene ključne riječi i indekse teksta na kojima su one pronađene.

Cijeli kodovi, kao i primjeri ulaznih datoteka i *readme* datoteka dostupni su na GitHub poslužitelju na sljedećoj poveznici: https://github.com/BioinformaticsFER/Aho-Corasick-algorithm.

3.1. UPUTE ZA INSTALACIJU

Sve tri implementacije pokreću se u sustavu Linux preko *Terminala* (f*ile1* je datoteka s ključnim riječima, a *file2* je datoteka s tekstom):

- Python
  - ► pozicioniraj se u direktorij gdje je smještena datoteka AhoCorasickAlgorithm.py
  - ► pokreni program sljedećom naredbom:

    $ python AhoCorasickAlgorithm.py file1 file2
- C#
  - ► raspakiraj komprimiranu datoteku AhoCorasick_C#.zip
  - ► instaliraj prevoditelj za C# sljedećom naredbom:

    $ sudo apt-get install mono-complete
  - ► pozicioniraj se u direktorij 'AhoCorasick_C#'
  - ► prevedi program sljedećom naredbom*:

    $ xbuild /p:OutputPath='desired_output_directory' AhoCorasick.sln
  - ► pokreni program sljedećom naredbom:

    $ mono AhoCorasick.exe file1 file2
  - ► *desired_output_directory = direktorij u kojem će se spremiti izvršna datoteka
- Java
  - ► raspakiraj komprimiranu datoteku AhoCorasick_Java.zip
  - ► pozicioniraj se u direktorij 'src'
  - ► prevedi program sljedećom naredbom:

    $ javac Start.java
  - ► pokreni program sljedećom naredbom:

    $ java Start file1 file2

4. REZULTATI

Algoritmi su testirani na šest sljedova nukleotida različitih duljina (19517-249250621 bp) iz *Ensembl* baze podataka [2]. Iz svakog kromosom odabrano je nekoliko ključnih riječi (podnizova) različitih duljina i na različitim pozicijama koje su se onda pretraživale u tom referentnom kromosomu. Svaka kombinacija pokretana je pet puta za svaku implementaciju kako bi se dobila što objektivnija srednja vrijednost (u dokumentaciji je zbog preglednosti dana samo srednja vrijednost). U tablicama je prikazana usporedba rezultata za sve kombinacije ulaza za tri različite implementacije za iduće parametre: vrijeme potrebno za konstrukciju automata, vrijeme potrebno za prolazak cijelog teksta kroz automat (u to vrijeme uključeno je i vrijeme učitavanja datoteke s tekstom) te količina potrošene memorije. U svakom stupcu je crvenom bojom obilježen najlošiji rezultat, a zelenom najbolji. Postotak točnosti je 100%, odnosno za svaki primjer svaka od implementacija uspješno je pronašla svaku ključnu riječ.

- primjer 1
  - datoteka teksta: Drosophila_melanogaster.BDGP5.74.dna.chromosome.dmel_mitochondrion_genome.fa
  - duljina teksta: 19 517
  - duljine ključnih riječi: 180, 240, 420, 600 $\rightarrow \Sigma|K_i| = 1440$

Tablica 3. Rezultati za primjer 1

	Konstrukcija DFA [ms]	Prolazak teksta kroz automat [ms]	Potrošena memorija [KiB]
Python	**5,014**	12,044	780
C#	11,96	**7,458**	**772**
Java	**21,84**	**36,272**	**2361**

- primjer 2
  - datoteka teksta: Drosophila_melanogaster.BDGP5.74.dna.chromosome.4.fa
  - duljina teksta: 1 351 857
  - duljine ključnih riječi: 600, 720, 900, 1200 $\rightarrow \Sigma|K_i| = 3420$

Tablica 4. Rezultati za primjer 2

	Konstrukcija DFA [ms]	Prolazak teksta kroz automat [ms]	Potrošena memorija [KiB]
Python	**10,332**	**672,932**	1072
C#	15,57	295,914	**996**
Java	**34,628**	**248,57**	**24374,4**

- primjer 3
  - datoteka teksta: Homo_sapiens.GRCh37.74.dna.chromosome.21.fa
  - duljina teksta: 48 129 895
  - duljine ključnih riječi: 600, 1200, 1440 $\rightarrow \Sigma|K_i| = 3240$

Tablica 5. Rezultati za primjer 3

	Konstrukcija DFA [ms]	Prolazak teksta kroz automat [ms]	Potrošena memorija [KiB]
Python	**9,214**	**18784,602**	1088
C#	13,014	7814,996	**1068**
Java	**34,396**	**1696,364**	**19488,4**

- primjer 4
  - datoteka teksta: Homo_sapiens.GRCh37.74.dna.chromosome.15.fa
  - duljina teksta: 102 531 392
  - duljine ključnih riječi: 720, 780, 1200, 1800 $\rightarrow \Sigma|K_i| = 4500$

Tablica 6. Rezultati za primjer 4

	Konstrukcija DFA [ms]	Prolazak teksta kroz automat [ms]	Potrošena memorija [KiB]
Python	**11,43**	**56999,762**	1360
C#	21,596	24348,56	**956,8**
Java	**42,802**	**5176,892**	**52720,8**

- primjer 5
  - datoteka teksta: Homo_sapiens.GRCh37.74.dna.chromosome.4.fa
  - duljina teksta: 191 154 276
  - duljine ključnih riječi: 300, 600, 900, 1200 $\rightarrow \Sigma|K_i| = 3000$

Tablica 7. Rezultati za primjer 5

	Konstrukcija DFA [ms]	Prolazak teksta kroz automat [ms]	Potrošena memorija [KiB]
Python	**7,93**	**109864,328**	1068
C#	18,148	47705,228	**916**
Java	**33,022**	**10440,356**	**17782,6**

- primjer 6
  - datoteka teksta: Homo_sapiens.GRCh37.74.dna.chromosome.1.fa
  - duljina teksta: 249 250 621
  - duljine ključnih riječi: 900, 1200, 1320 $\rightarrow \Sigma|K_i| = 3420$

Tablica 8. Rezultati za primjer 6

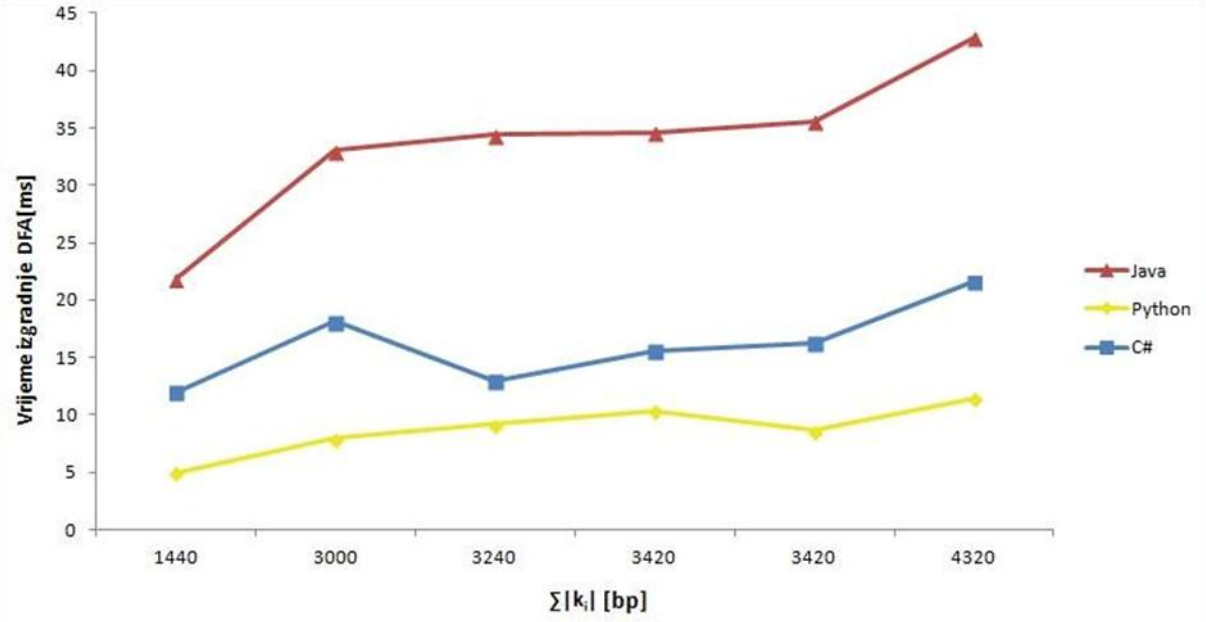
	Konstrukcija DFA [ms]	Prolazak teksta kroz automat [ms]	Potrošena memorija [KiB]
Python	**8,638**	**120253,466**	1088
C#	16,238	50039,754	**996**
Java	**35,57**	**10625,17**	**33122,2**

5. ZAKLJUČAK

Aho-Corasick algoritam pokazao se izuzetno učinkovit u pretrazi više ključnih riječi odjednom u dugačkim sekvencama DNA. Izgradnja automata za sve primjere bila je ispod 45 ms, a pretraga cijelog teksta kretala se od 7,5 ms do 2 min .

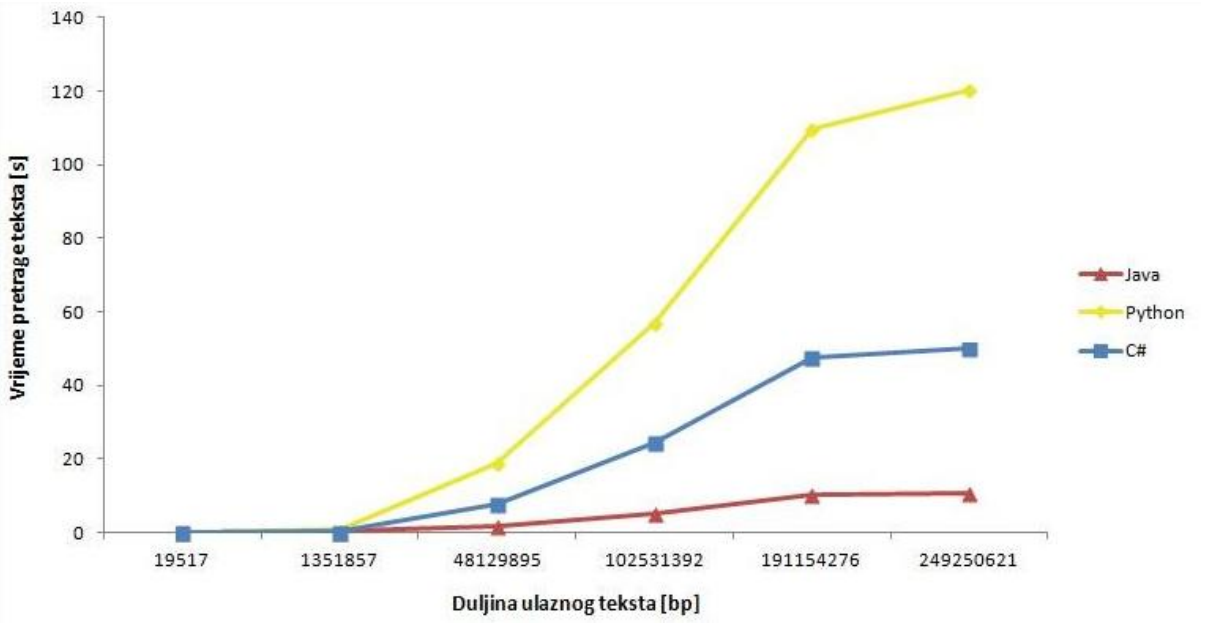
U ovome poglavlju dat će se zaključak o utjecaju veličina ključnih riječi i njihovog broja te duljine teksta na tri mjerena parametra: vrijeme potrebno za izgradnju DFA, vrijeme potrebno za pretragu cijeloga teksta i zauzeće memorije.

Vrijeme potrebno za izgradnju DFA ne ovisi o duljini samog teksta, nego o broju ključnih riječi, odnosno o njihovim duljinama. Najbolje vrijeme za svaki primjer postizalo se u Pythonu, dok je najlošije vrijeme za sve primjere bilo u Javi. U prosjeku je Python bio dva puta brži od C#, a četiri puta brži od Jave. Grafički prikaz vremena izgradnje DFA u ovisnosti u zbroju duljina ključnih riječi prikazan je na slici 8.



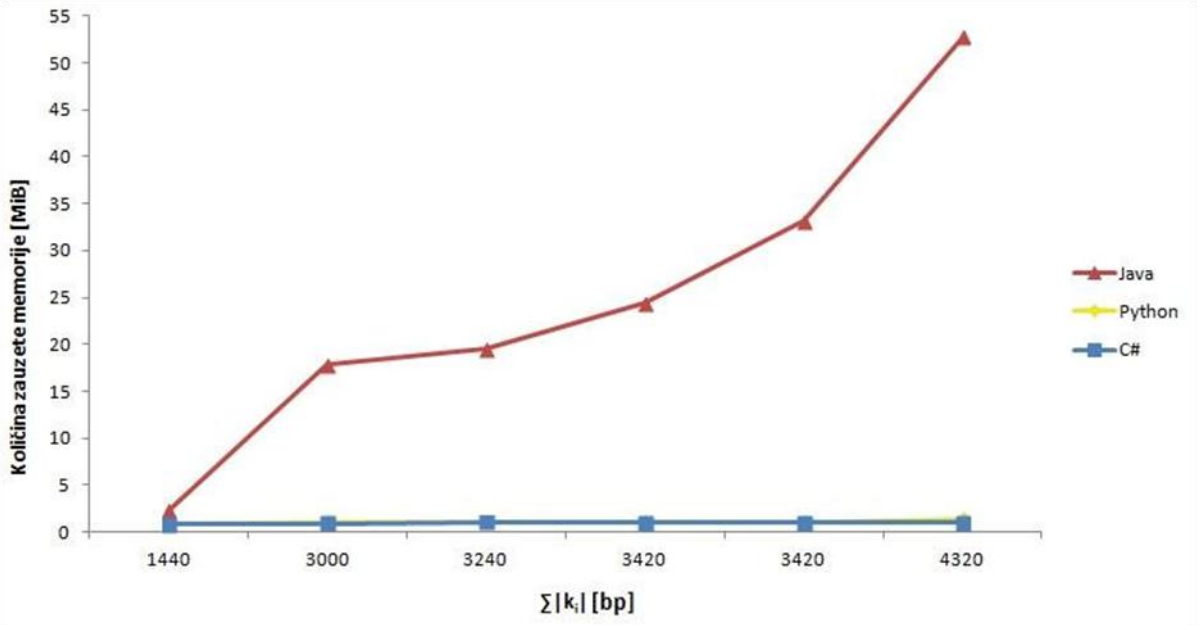
Slika 8. Vrijeme izgradnje DFA

Vrijeme potrebno za pretragu teksta ovisi uglavnom o duljini teksta. Najbolje vrijeme za sve primjere osim za najmanji bilo je u Javi, dok je za najmanji primjer izvedba Jave bila najsporija, a C# najbrža. Izvedba u Pythonu je za sve primjere, osim najmanjeg, bila najsporija. U prosjeku je Java bila deset puta brža od Pythona, a pet puta brža od C#. Grafički prikaz vremena u ovisnosti u duljini teksta prikazana je na slici 9.



Slika 9. Vrijeme pretrage teksta

Količina zauzete memorije je zbog same implementacije značajno više ovisila o ključnim riječima nego o duljini teksta. U sve tri implementacije iz teksta se slijedno učitava linija po linija teksta te se ona provlači kroz automat. Zbog toga memorija ovisi samo o duljini jedne linije teksta. Nasuprot tome, količina zauzete memorije raste s povećanjem broja ključnih riječi, odnosno njihovih duljina. Za sve primjere Java je imala najveće zauzeće memorije, dok su Python i C# pokazivali gotovo jednake rezultate, u prosjeku ≈10 puta manje zauzeće memorije od Jave. Unatoč tome, sve tri implementacije pokazuju dobar utrošak memorije jer je najlošiji rezultat 160MiB. Zbog većeg utjecaja ključnih riječi na memoriju, na slici 10. je grafički prikazana ovisnost memorije o zbroju duljina ključnih riječi.



Slika 10. Količina zauzete memorije

6. LITERATURA

[1] Alfred V. Aho, Margaret J. Corasick, Efficient string matching: an aid to bibliographics search. Communications of the ACM, izdanje 6, lipanj 1975., str. 333 – 340, ACM New York, NY, USA ftp://163.13.200.222/assistant/bearhero/prog/%A8%E4%A5%A6/ac_bm.pdf, datum pristupa stranici: prosinac 2013.

[2] Ensembl, http://www.ensembl.org/index.html, datum pristupa stranici: prosinac 2013.