SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

PROJEKT IZ PREDMETA BIOINFORMATIKA

Smith-Waterman s linearnom memorijskom složenošću

Vlatka Pavišić, Janko Sladović, Andrija Stepić, Marko Vrljičak

Zagreb, siječanj, 2014

Sadržaj

1. Uvod

Poravnavanje sekvenci je jedan od središnjih problema bioinformatike. Cilj je dva ili više slijeda DNK ili proteina optimalno poravnati uz minimalan trošak transformacije. Rezultati mogu ukazivati na funkcionalne, strukturalne ili evolucijske veze između sekvenci.[2]

Dvije su vrste poravnanja - globalno i lokalno. Globalno poravnanje se proteže duž svih sekvenci, dok lokalno dopušta neporavnate znakove na početku i/li na kraju sekvenci i tako detektira regije najveće sličnosti.

Prvu generaciju algoritama za pronalazak poravnanja dvije sekvence čine metode dinamičkog programiranja. Njih karakterizira sporije izvođenje u odnosu na modernije heurističke metode, no one garantiraju točnost. Smith-Waterman algoritam nalazi optimalno lokalno poravnanje dvije sekvence i ima kvadratnu vremesku i memorijsku složenost. S ciljem ubrzanja i uštede na memoriji razvijeno je nekoliko njegovih prerada, a jedna od njih je rekurzivni Hirschbergov algoritam. Potonji se odlikuje linearnom memorijskom složenošću, što je od velike važnosti u stvarnim primjenama gdje sekvence poprimaju velike dimenzije. Needleman-Wunsch algoritam se koristi za globalna poravnanja, a Smith-Waterman je njegova varijacija. Kombinacijom ova tri algoritma moguće je ostvariti lokalno poravnanje dvije sekvence sa linearnom memorijskom složenošću.U radu će biti opisana ideja rješenja te će biti uspoređene četiri različite implementacije.

2. Algoritam

2.1. Smith-Waterman

Algoritam Smith-Waterman pronalazi optimalno lokalno poravnanje dvije sekvence. Uzmimo za primjer dvije sekvence:

x = GGCTCAATCA

y = ACCTAAGG

Njihovo lokalno poravnanje je

CTCAA

CT- AA

U "izrezanim" regijama sekvence x i y su najsličnije. Na mjestima gdje se ne podudaraju umeću se praznine (engl. *gap*). Poravnanje je optimalno ako minimizira trošak umetanja, zamjene i brisanja potrebnih da se jedan niz transformira u drugi.

Prije izvođenja algoritma moraju se definirati konstante kojima se ocjenjuje podudaranje (*match*), nepodudaranje (*mismatch*) i otvaranje praznine (*gap*).

Potom je potrebno konstruirati matricu F dimenzija (n+1)x(m+1), gdje je n duljina niza x, a m duljina niza y. Prvi red i prvi stupac popune se nulama:

$$F[i,0] = 0, 0 < i < n,$$

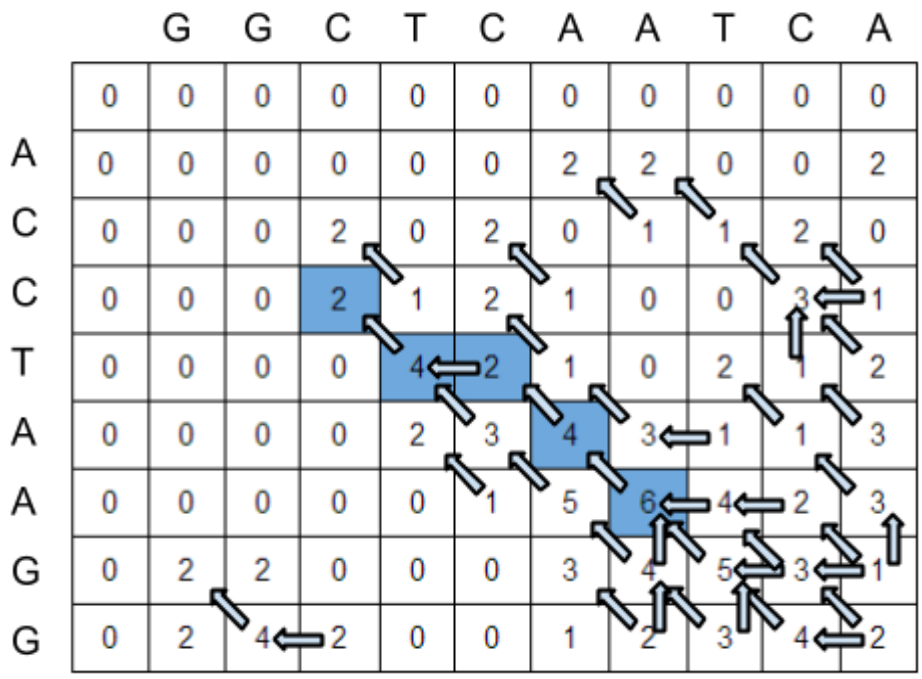
$$F[0,j] = 0, 0 < j < m$$

Ostali elementi se, počevši sa F[1,1], redom popunjavaju prema pravilu

$$F[i,j] = Max(0, F[i-1,j] + MatchCost(s1[i-1], s2[j-1]),$$

$$F[i-1,j] + GapCost, F[i,j-1] + GapCost)$$

Svaka ćelija matrice F[i,j] tako sadrži vrijednost podudaranja prefiksa x[1...i] sa prefiksom y[1...j]. Također se pamti ćelija iz koje se došlo, radi rekonstrukcije puta. Primjer popunjavanja tablice za navedena dva niza prikazan je na slici 1.



Slika 1 - Provođenje Smith-Waterman algoritma za match = 2, mismatch = -1 i gap = -2

Lokalno poravnanje se dobiva rekonstrukcijom puta počevši od maksimalne vrijednosti u tablici i slijedeći strelice do zadnje pozitivne vrijednosti u tablici. U slučajevima kada je moguće više prijelaza preferira se dijagonalni put. Zatim se poravnanja očitavaju iz označenog puta. Dijagonalni prijelaz označava podudaranje, horizontalni umetanje praznine u niz x (*insertion*), a vertikalni brisanje (*deletion*) iz niza x (umetanje praznine u niz y).

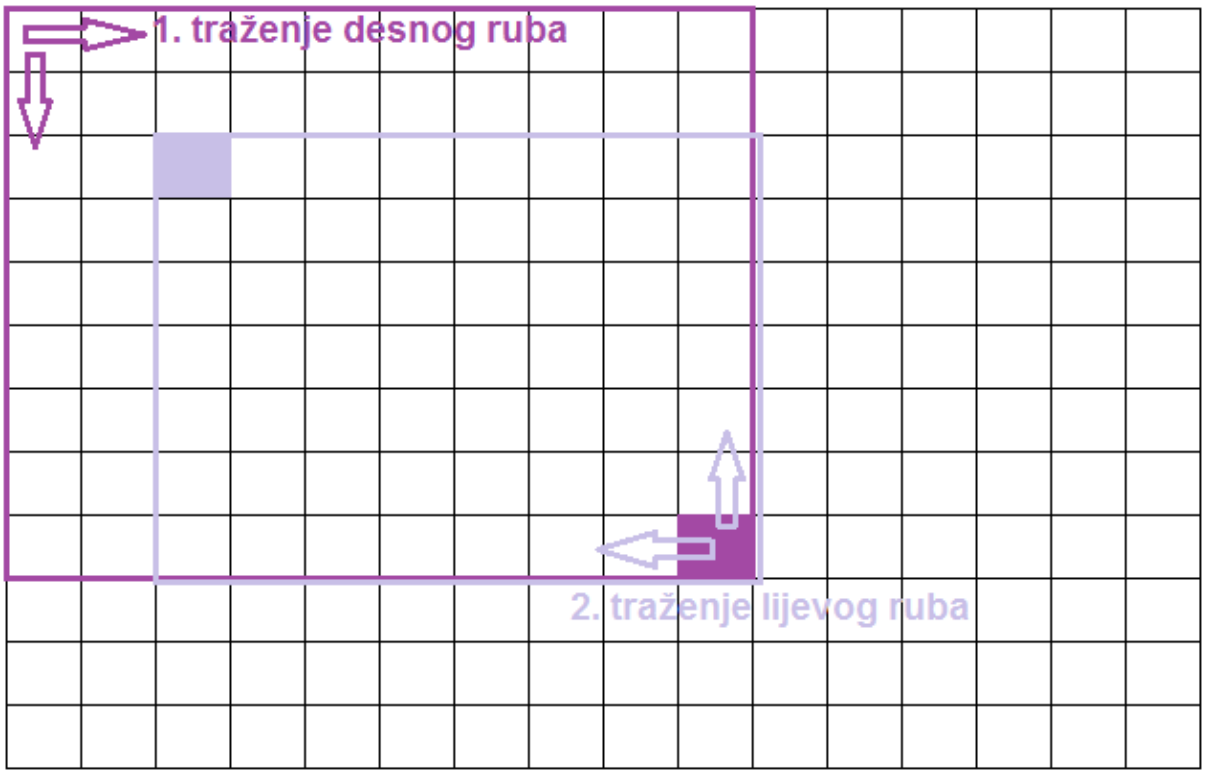
Memorijska složenost ovog algoritma je O(nm) jer je potrebno pamtiti cijelu matricu radi rekonstrukcije puta. Samo računanje njenih ćelija može se izvesti sa dva reda matrice koja se pomiču prema dolje. Naime, za računanje vrijednosti jedne ćelije potrebne su samo vrijednosti ćelija lijevo od nje, iznad nje i dijagonalno lijevo iznad nje. Taj trik se koristi u ovoj implementaciji.

Cilj je ograničiti područja optimalnog lokalnog podudaranja i u njima naći globalno poravnanje algoritmom koji ima linearnu memorijsku složenost - Hirschbergovim algoritmom, koji će biti opisan kasnije.

Desna granica lokalnog poravnanja nalazi se Smith-Watermanovim algoritmom koji pamti samo dva trenutna reda matrice - za računanje F[i,j] pamte se (i-1) i i-ti red, maksimalna izračunata vrijednost ćelije i njezina pozicija (indeksi i_max i j_max). Nizovi x i y

se zatim skrate tako da završavaju sa regijama optimalnog lokalnog poravnanja (x do indeksa i_max, y do indeksa j_max).

Lokalno poravnanje se treba ograničiti i s lijeva, pa se opisani postupak ponavlja nad obrnutim nizovima dobivenima iz prethodnog koraka, kao što prikazuje slika 2.



Slika 2 - Svođenje problema lokalnog poravnanja na problem globalnog poravnanja

Tako dobivena dva podniza označavaju područja optimalnog poravnanja početnih nizova x i y. No, to poravnanje nije rekonstruirano te se ne zna gdje treba umetnuti praznine. Problem se sada može sagledati globalno i na njega primjeniti algoritam globalnog poravnavanja s linearnom memorijskom složenošću - Hirschbergov algoritam.

2.2 Hirschberg

Hirschbergov algoritam je nastao iz Needleman-Wunsch algoritma. On je sličan opisanom Smith-Watermanu, a razlikuje se u nekoliko aspekata:

1. pronalazi globalno, a ne lokalno poravnanje

2. početni uvjeti nisu jednaki nuli, nego ovise o cijeni praznine
3. prilikom računanja vrijednosti u ćelijama se minimizira
4. rekonstrukcija počinje od zadnje ćelije u matrici

Navedene razlike ne utječu na složenost algoritma, pa i originalni Needleman-Wunsch ima kvadratnumemorijskusloženost. Verzija Dana Hirschberga ima kvadratnu vremensku složenost, no memorijsku složenost reducira na O(max(n,m)).

Hirschbergov algoritam radi na principu “podijeli pa vladaj”. Kako bi se izbjegla kvadratna memorijska složenost, Hirschbergov algoritam u svakom trenutku pamti najviše dva reda tablice Needleman-Wunschovog algoritma.

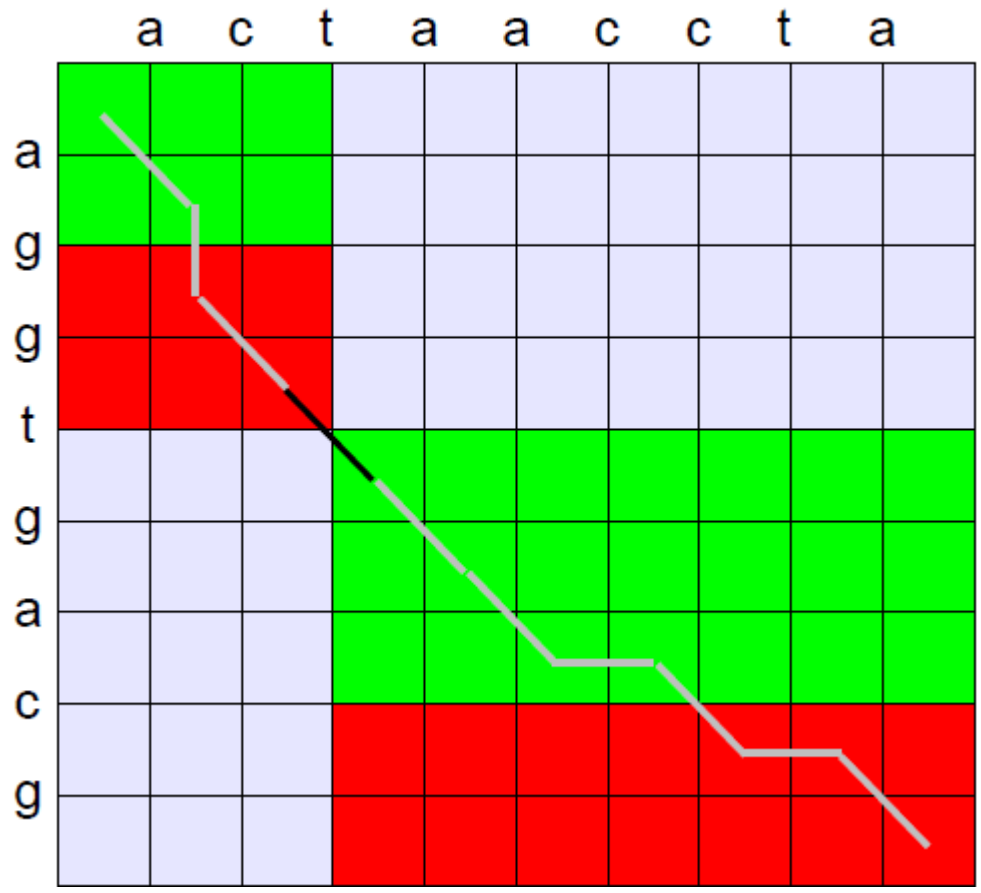
Hirschbergov algoritam je rekurzivan, računa se središnji brid, što će biti opisano kasnije, te se potom oko središnjeg brida stvaraju dva podniza za svaku od ulaznih vrijednosti, gornji lijevi te donji desni, a s njima se ponovno poziva isti postupak.

Isti postupak se ponavlja dok ne dođemo do barem jednog podniza s duljinom jedan ili nula, u prvom slučaju onda možemo primjeniti običan Needleman-Wunschov algoritam pošto će memorijska složenost opet biti linearna, dok u drugom slučaju jednostavno završavamo rekurziju te poravnavamo znakove u nizu čija je vrijednost različita od nule s praznim mjestima u nizu čija je vrijednost jednaka nuli.

Računanje središnjeg brida odvija se pomoću dva središnja reda tablice. Odredimo red koji je najčešće polovica vrijednost duljine podniza koji čini stupce matrice te dijelimo taj podniz na dva kraća podniza za svaki od kojih tražimo poslijednji red tablice Needleman-Wunschovog postupka. Kod traženje donjeg reda, potrebno je paziti da koristimo obrnuta oba podniza, a kod traženja središnjeg brida ponovo moramo obrnuti donji red tablice.

Nakon što je određen gornji i donji središnji red tablice, potrebno je naći brid tako da se nađe onaj indeks u retcima gdje je suma gornjeg i donjeg elementa najveća. Tako smo našli središnji brid te oko njega razdvajamo podniz koji čini retke originalne matrice te pozivamo rekurziju s gornjim i lijevim, odnostno donjim i desnim, podnizovima.

Na slici 3 je prikazan primjer provođenja Hirschbergovog postupa u trenutku pozivanja drugog koraka rekurzije, zeleni i crveni dijelovi predstavljaju podnizove koji će biti korišteni za izračunavanje gornjeg odnosno donjeg srednjeg stupca tablice, dio konačnog poravnanja obojan u crno predstavlja središnji brid, a dijelovi obojani u bijelo su oni podnizovi koji neće biti razmatrani u idućem koraku.



Slika 3 - Primjer računanja poravnanja pomoću Hirschbergovog algoritma

Glavna prednost Hirschbergovog algoritma nad Smith-Watermanovim ili standardnim Needlman-Wunschevim algoritmom jest u tome što Hirschbergov algoritam u pravilu ne razmatra vrijednosti svih elemenata tablice, već se fokusira na one bliže optimalnom poravnanju.

3. **Implementacije i testiranje**

Za potrebu ovog projekta, izrađene su četiri različite implementacije Smith-Watermanovog algoritma s linearnom memorijskom složenošću. Korišteni programski jezici su bili C# (Andrija Stepić), Java (Janko Sladović), Perl (Marko Vrljičak) te Python (Vlatka Pavišić).

U sve četiri verzije program čita genom iz ulaznih datoteka čije ime zadaje korisnik, poziva potrebne algoritme te na kraju ispisuje dobiveno optimalno poravnanje u zasebnu izlaznu datoteku, čije ime te po potrebi lokaciju također zadaje korisnik.

Testiranja su bila vršena nad sve četiri implementacije koristeći nasumično generirane nizove različitih duljina. Nizovi su bili zapisani u FASTA formatu, koji su sve implementacije algoritma morale moći čitati te su bili sastavljeni od nasumičnog niza znakova A, C, G i T. Najkraći niz nad kojim su vršena testiranja je imao 100 takvih znakova, a također su korišteni nizovi od po 1000, 10000, 100000 i milijun znakova.

Primjer jednog od korištenih nizova sa 100 znakova je:

```
TAGACTGTCAACGACAAACATCCGTCCTTCCTATCAACACCTCGATCCATGCCCGATCCTCACGATGAAACATTA
GTAGCTAACCTTCTGTGCCCCCAAC
```

Primjer jednog od nizova sa 1000 znakova:

```
CTGACCAGTTGGAATAAGCATGCGATCTTAACTATCAAAACGTTTTAAGAAAAGGTCATATAAGCCGTACAATGA
CAGAGTCCGGAGATTCTTGTGTTGGAAACATAGTAGTATAATACCTACGAGTTGTGCCCCTAAGTCCAGGGCTGA
GTACGGAAAATTGTACTAGTACGCGTTGATATCCCCTAGGAGCCTTCATGTGTGGACCTCATCAGGAAGACGAAG
CTAAGTTGGTATAGGGTCCCACAGCCACCCTTCAATTGACAGCTAAGGCATTTCAGGGGTACCCGCAGATATAAA
TTGTCATAGCGAACCCCAAGCTGACCATCTGGTACTGACGCGCTCTGAGCGAGCACATAACATGTACTTGCTCGA
AACTGTTTTGTTGCCCTGCGGTTCCAGCTGCATGCGTTAGACAAGAATAGTCCACTCTTCTGTATTCGAACGCGA
CATCACCGTTCTAATGATCCGGGAAGTTAAATGCATACCTCAAGACGAGTTTCACTTATCGAGAGTCAGCAACCC
CTCTCGCATCTATCTGACGTTGCCCCGCGTGAAAGCATGTAGGAGAGACCAGCCATCCAATGGCATCTCCTTCCT
CGGTTACTACTCGAAAGCCCCTATGTTCGTAAGAAACCACGCGGAAACAAGCACTTAGTGAAAGGCTGCTCGCTA
TCTCCCCATATTCGACATCCAGAAGAAGCTCCGTCGCCGGCTTCTGTCTACATATCGTGTGGTACGGCGTCTATT
AGAACATGGGTCACAAGTGGCGTGAGGGATGTCCAGAAGCAGGCGAGTCTAGTCTACATACACACAGAAGTGACA
GCCTCAATGAAGTGGTCACAAAGGCGATCTTCGAATAGTAATGGTCGAGACATGTTCCTGCGCGTAGTTCTACAC
GATAAAGAGCTCACAGAGACTAAAAGCGGCGGTGGTAACAAGGGTGGTTATTAAGGTTAGTCCACAGACACCTAC
CAGCACGCACCCGGGTAACCCTTGA
```

Jezik					
Duljina prve sekvence	Duljina druge sekvence	C#	Java	Python	Perl
10^2	10^2	0.01842 s	0.06656 s	0.06025 s	0.09196 s
10^3	10^2	0.02149 s	0.10451 s	0.14958 s	0.43023 s
10^3	10^3	0.09987 s	0.32992 s	3.54217 s	9.48722 s
10^4	10^3	0.36812 s	0.63270 s	17.39963 s	46.447793
10^4	10^4	6.14174 s	12.75127 s	427.21399 s	1002.60669 s
10^5	10^5	630.68986 s	1327.45527 s	832 min 42 s	1731 min 19 s
10^6	10^6	1157 min 12 s	2555 min 27 s	-	-

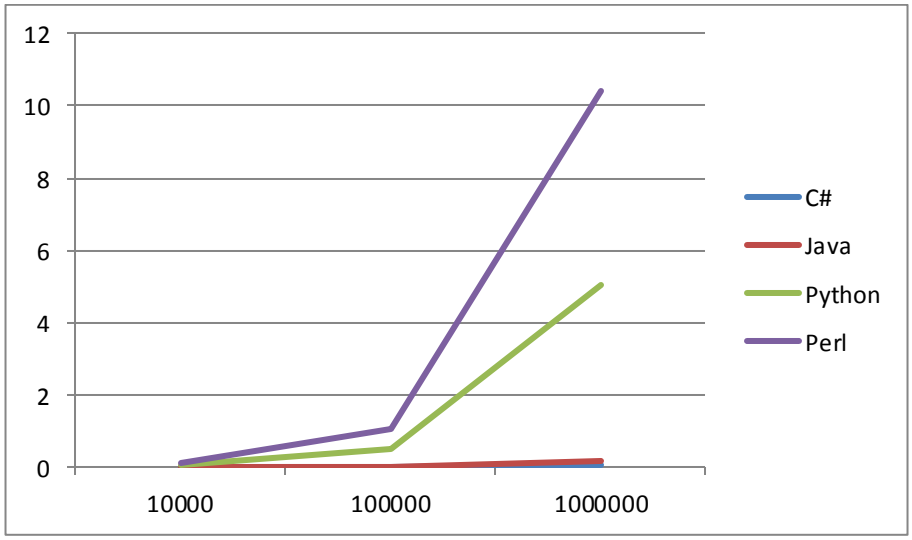
Tablica 1 – prosječna duljina izvođenja algoritma

Jezik					
Duljina prve sekvence	Duljina druge sekvence	C#	Java	Python	Perl
10^2	10^2	6632 kB	18856 kB	5908 kB	3104 kB
10^3	10^2	6680 kB	19992 kB	5908 kB	3104 kB
10^3	10^3	6840 kB	21880 kB	5908 kB	3632 kB
10^4	10^3	7146 kB	22508 kB	5908 kB	3632 kB
10^4	10^4	8096 kB	27984 kB	6960 kB	4164 kB
10^5	10^5	13320 kB	33068 kB	14296 kB	13932 kB
10^6	10^6	67540 kB	78644 kB	86680* kB	113128* kB

Tablica 2 – prosječno zauzeće memorije pri izvođenju algoritma

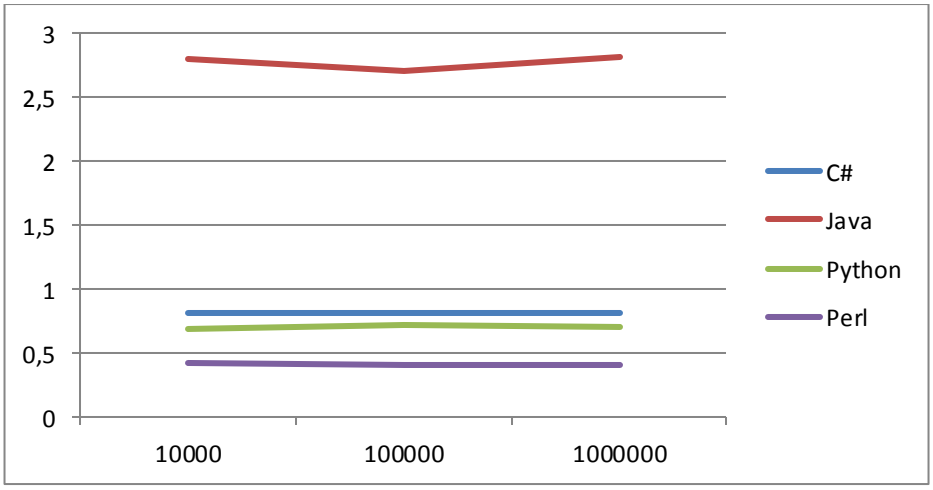
Testiranja su izvršena na operacijskom sustavu Biolinux 7, koji se nalazio na virtualnom stroju s 1 GB RAM, 2.53 GHz taktom procesora. Rezultati u tablicama su prosjek od nekoliko pokretanja za svaki primjer (10 za manje primjere, 1-2 za najveće). Posao je u svim implementacijama slično vremenski podijeljen - prelazak iz lokalnog u globalni slučaj uzima 55% vremena, Hirschbergov algoritam uzima 45% (u stvarnom slučaju ostale funkcije, nevezane uz sam algoritam, uzimati će do 5% vremena).

Iz testiranja se može primjetiti da vrijeme izvođenja raste sa približno kvadratnom, a memorija linearnom složenošću u ovisnosti o duljini ulaznih sekvenci, kao što je navedeno u teorijskoj raspravi Hirschberg algoritma. Točnost algoritma je, naravno, 100%, jer ovaj algoritam osmišljen i implementiran tako da uvijek nađe optimalno rješenje.



Graf 1 – prikaz ovisnosti vremena izvođenja algoritma o duljini nizova

Prikaz na grafu 1 dobiven je dijeljenjem vremena sa duljinom niza, te je iz grafa vidljiva kvadratna ovisnost. Početne vrijednosti za nizove duljine 100 i 1000 odudaraju od linearnosti jer za tako malene nizove vrijeme koje odlazi na inicijalizacije i čitanje datoteka je usporedivo sa vremenom za na obradu niza, pa ta vremena nisu mjerodavna za prikazanu ovisnost.



Graf 2 – prikaz ovisnosti memorije potrebne za izvođenje algoritma, mjerene u kB, o duljini nizova

Prikaz na grafu 2 dobiven je dijeljenjem iznosa potrebne memorije sa duljinom niza, iz čega se može primjetiti da je memorijska ovisnost o duljini linearna. Prikazi zauzeća memorije za duljine nizova 100 i 1000 su izostavljeni jer su njihovi omjeri previše odudarali od ostalih, budući da je za te duljine zauzeće potrebno za biblioteke i pomoćne varijable sumjerljivo memoriji za same nizove.

4. **Zaključak**

Smith-Watermanov algoritam za lokalno poravnanje je dobar način za rješavanje problema poravnavanja genoma, ali problem je što s povećanjem duljine genoma dolazi do pretjeranih memorijskih zahtjeva. Kako bi se izbjegli ti memorijski zahtjevi, koristi se Hirschbergov algoritam za poravnavanje nizova koji ne zahtjeva pamćenje cijele tablice u svojoj memoriji. Na dobivenim rezultatima se može vidjeti da za nijednu od implementacija memorijsko zauzeće pri poravnanju nizova od desetak tisuća te stotinjak tisuća znakova ne prelazi u neke neprihvatljive iznose. Također, Hirschbergov algoritam zbog porasta u memorijskoj efikasnosti ne žrtvuje pretjerano samu brzinu izvođenja, koja je i dalje kvadratna, iako možda malo sporija od originalne ideje Smith-Watermanovog algoritma.

5. Literatura

[1] Uvod u dinamičko programiranje, te teorija Smith-Waterman, Needleman-Wunsch i Hirschberg algoritma, http://www.fer.unizg.hr/_download/repository/Bioinformatika_Dinamicko_programiranje_2.pdf, 16.11.2013.

[2] Teorija o poravnanju nizova,http://en.wikipedia.org/wiki/Sequence_alignment, 16.11.2013.

[3] Teorija i pseudokod (nepotpuni) za Hirschbergov algoritam, http://en.wikipedia.org/wiki/Hirschberg's_algorithm, 16.11.2013.

[4] Dodatna teorija za Hirschbergov algoritam, http://www.science.marshall.edu/murraye/Clearer%20Matrix%20slide%20show.pdf, 16.11.2013.

[5] Prezentacija sa objašnjenjem rada Hirschbergovog algoritma uz pseudokod i napomene za implementaciju, http://users-cs.au.dk/cstorm/courses/AiBS_e12/slides/LinearSpace.pdf, 16.11.2013.

[6] Java aplikacija sa implementacijom Needleman-Wunsch i Smith-Waterman algoritma te ispisom izračuna korak po korak, korištena za usporedbu i validaciju rješenja prilikom implementacije, http://baba.sourceforge.net/, 16.11.2013.