SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

BIOINFORMATIKA - PROJEKT

essaMEM: finding maximal exact matches using enhanced sparse suffix arrays

Daniel Guja, Antun Razum, Petra Rebernjak
Voditelj: *doc. dr. sc. Mirjana Domazet-Lošo*

Zagreb, siječanj 2017.

SADRŽAJ

1. Uvod

Komparativna genomika je područje bioinformatike u kojem se uspoređuju značajke genoma različitih organizama. U ovoj grani bioinformatike, uspoređuju se dijelovi ili pak cijeli genomi kako bi se pronašle i proučavale sličnosti i razlike. Glavni princip komparativne genomike je činjenica da su zajedničke značajke organizama zapisane u DNK te su one evolucijski sačuvane.

Početak komparativne genomike je vrijeme kada su postali dostupni cijeli genomi dvaju organizama. To su genomi bakterija *Haemophilus influenzae* i *Mycoplasma genitalium* 1995. godine. Komparativna genomika je danas standardni alat u analizi svih novih slijedova genoma. Ovo područje bioinformatike je otkrilo velike sličnosti između nekih organizama kao što su ljudi i čimpanze. Također, pokazala je i velike razlike u sastavu gena u različitim evolucijskim lozama.

Rastom sve većeg broja dostupnih genoma javlja se potreba za novim tehnologijama u usporedbi nizova. Kombiniranjem i slaganjem genoma pomoću tih tehnologija i usporedba dobivenih usko povezanih genoma biti će vrlo značajna u otkrivanju novih znanja, od otkrivanja gena i velikih genomskih reorganizacija za utvrđivanje razlika između srodnih vrsta do otkrivanja novih lijekova za do sada neizlječive bolesti.

Jedan od alata bitnih za daljnji razvoj je izračunavanje maksimalnih točnih slaganja između dvaju nizova. Izračunavanje maksimalnih točnih slaganja između nizova bitno je za sljedeće: izračunavanje i rangiranje sličnosti genoma te utvrđivanja ključnih točaka u uspoređivanju genom-genom.

Cilj ovog projektnog zadatka je proučiti *essaMEM*[4], složeni algoritam za efikasno računanje maksimalnih točnih slaganja za velike nizove te implementirati podskup funkcionalnosti koje su autori objasnili u radu, konkretno kostrukciju *child* polja.

U idućem poglavlju dan je kratki pregled *essaMEM* algoritma. U poglavlju 3 opisano je *child* polje i njegova konstrkcija. U poglavlju 4 opisan je način vrednovanja rješenja, prikazani su rezultati izvršavanja te usporedba s referentim rješenjem. Konačno, u poglavlju 5 donesen je zaključak rada.

2. Pronalazak maksimalnih točnih slaganja

U idućem odjeljku ovog poglavlja opisan je razvoj ideje o pronalasku maksimalnih točnih slaganja i postanak essaMEM algoritma. U odjeklju 2.2 ukratko je opisan sam algoritam.

2.1. Razvoj algoritma

Maksimalno točno slaganje (engl. *Maximal Exact Match, MEM*) je podniz oba niza koji predstavlja točno slaganje između njih koje se ne može produžiti lijevo niti desno bez da se dogodi neslaganje između dva niza koja se uspoređuju. Za pronalaženje MEM-ova potrebna je pogodna struktura podataka. Jedna od njih je sufiksno stablo. Sufiksno stablo je osnovna struktura podataka koja omogućuje brzu analizu nizova. Sufiksno stablo ima široku primjenu u bioinformatici. Mnoštvo primjena opisano je posebno početkom 21. stoljeća. Kako je potreba za korištenjem sufiksnih stabala rasla, bilo je potrebno pronaći novu strukturu podataka koja će unaprijediti već postojeću. Tako se došlo do alternativne strukture podataka, sufiksnog polja (engl. *Suffix Array, SA*). Sufiksno polje je zamijelo sufiksno stablo. Glavni razlog tome je memorijski utrošak sufiksnog stabla. Sufiksno stablo za ulazni niz od n znakova zahtijeva najmanje $10n$ okteta memorijskog prostora u praksi, a često i $15n$ - $20n$ okteta. Sufiksno polje teorijski zauzima $O(nlogn)$ bita, a u praksi obično $4n$ okteta.

Uobičajeno se koriste sufiksna stabla ili unaprijeđena sufiksna polja (engl. *Enhanced Suffix Arrays, ESA*) za pronalaženje MEM-ova. ESA se sastoji od četiri polja: sufiksnog polja, polja najvećih zajedničkih prefiksa (engl. *Longest Common Prefix, LCP*), child polja i polja sufiksnih poveznica (engl. *Suffix Link Arrays*) koja sadrže dijelove informacija iz sufiksnog stabla te zajedno dostižu potpunu ekspresivnost sufiksnih stabala. Dakle, svaki problem koji se može riješiti korištenjem sufiksnih stabala, može se riješiti i korištenjem ESA s istom asimptotskom složenošću.

Khan *et. al.* u [3] savjetuje korištenje rijetkih sufiksnih polja (engl. *Sparse Suffix Arrays, SSA*). Kod SSA, indeksira se svaki *K*-ti sufiks. Parametar *K* naziva se parametar rijetkosti, (engl. *sparsness factor*). Algoritam koji koriste autori u radu [3] za pronalaženje MEM-ova temelji se na SSA i pronalazi MEM-ove puno brže od prethodnih metoda. Uz brže pronalaženje MEM-ova algoritam koristi i značajno manje memorije od prethodnih metoda. Posljedično, uporabom SSA moguće je indeksirati znatno veće genome.

Vyverman *et. al.* u [4] želi poboljšati metodu predstavljenu u [3] implementiranjem rijetkih child polja za velike parametre rijetkosti. Dodatno, korištenjem sufiksnih poveznica (engl. *suffix links*) i child polja gradi se unaprijeđeno rijetko sufiksno polje (engl. *Enhanced Sparse Suffix Array, ESSA*) koje ima istu ekspresivnost kao i sufiksno stablo za podnizove veće od *K*. Memorijski učinkoviti algoritmi za pronalaženje MEM-ova mogu se podijeliti u dvije kategorije: *online* i *index* metode. essaMEM pripada kategoriji indeksnih metoda algoritama za pronalaženje MEM-ova. Ova metoda uspoređuje niz Q veličine m s indeksnom strukturom izgrađenom nad referentnim nizom S veličine n. Prednost ovog pristupa je ta što se konstruirana indeksna struktura može ponovno iskoristiti, odnosno za sljedeću usporedbu nizova nije potrebno ponovno indeksirati referentni niz.

2.2. essaMEM algoritam

essaMEM algoritam sastoji se od nekoliko odvojenih koraka koji se izvršavaju na sljedeći način. Prvo se konstruira SA na način da se pozove *radix sort* za sortiranje sufiksa prema njihovom prefiksu određene duljine, a zatim se pozove funkcija za sortiranje sufiksa `qsufsort` implementacije, koja dovršava sortiranje. Slijedi konstrukcija inverznog sufiksnog polja (engl. *Inverse Suffix Array, ISA*) koje se direktno konstruira iz SA. Nakon toga konstruira se LCP polje upotrebom algoritma iz [2]. Za njegovu konstrukciju potrebni su SA i ISA. Nakon ovih predkoraka može se konstruirati child polje. Za to je potrebno samo LCP polje.

Navedene strukture su dovolje za ispravan i brz rad algoritma. U ostatku originalnog algoritma pretraga MEM-ova vrši se prolaskom po navedenim strukturama, no to izlazi izvan okvira ovog projekta.

3. Child polje

U ovom poglavlju opisan je child niz, glavna značajka ovog algoritma. U idućem odjeljku donesen je kratak uvod. U odjeljku 3.2 ukratko je opisan način konstrukcije child niza.

3.1. Uvod i definicije

MEM-ovi se pronalaze obilaskom sufiksnog stabla odozgo prema dolje. Koristeći SA i LCP polje ovaj problem moguće je riješiti u složenosti $O(m + logn)$. Ukoliko se SA proširi dodatnim child poljem problem je moguće riješiti u složenosti $O(m)$.

Dohvat djece unutarnjih čvorova konceptualnog sufiksnog stabla ostvarenog pomoću SA i LCP polja moguće je obaviti u konstantnom vremenu koristeći child polje. Za izgradnju child polja koristi se informacija o LCP intervalima, definiranima u [1]. Bitno je naglasiti da su LCP intervali samo konceptualni te se nikada ne konstruiraju niti čuvaju u memoriji.

Child polje je duljine n/K, a svaki element u polju sadrži vrijednosti *up*, *down* i *nextLIndex*. Te vrijednosti definirane su u nastavku:

$$childtab[i].up = min\{q \in [0..i-1] \mid lcptab[q] > lcptab[i] \text{ i } \forall k \in [q+1..i-1] : lcptab[k] \geq lcptab[q]\}$$

$$childtab[i].down = max\{q \in [i+1..n] \mid lcptab[q] > lcptab[i] \text{ i } \forall k \in [i+1..q-1] : lcptab[k] > lcptab[q]\}$$

$$childtab[i].nextLIndex = min\{q \in [i+1..n] \mid lcptab[q] = lcptab[i] \text{ i } \forall k \in [i+1..q-1] : lcptab[k] > lcptab[i]\}$$

Tablice 3.1 i 3.2 prikazuju vrijednosti sufiksnog i LCP polja te sve tri vrijednosti child polja za niz znakova *mississippi* i za vrijednosti parametra $K = 1$ odnosno $K = 2$.

Child polje nosi informaciju o odnosu roditelj-dijete u LCP intervalima. Za LCP interval [i..j] s *l*-indeksima $i_1 < i_2 < ... < i_k$, $childtab[i].down$ ili $childtab[j + 1].up$ vrijednost koristi se za dobivanje prvog *l*-indeksa i_1. Ostali *l*-indeksi i_2, ..., i_k

Tablica 3.1: SA niza S = `mississippi` proširen LCP i child tablicom ($K = 1$).

i	sa[i]	lcp[i]	up[i]	down[i]	nextLIndex[i]	s[sa[i]]
0	11	0			1	$
1	10	0		2	5	i$
2	7	1			3	ippi$
3	4	1		4		issippi$
4	1	4				ississippi$
5	0	0	2		6	mississippi$
6	9	0		7	8	pi$
7	8	1				ppi$
8	6	0	7	10		sippi$
9	3	2				sissippi$
10	5	1	9	11		ssippi$
11	2	3				ssissippi$

Tablica 3.2: SA niza S = `mississippi` proširen LCP i child tablicom ($K = 2$).

i	sa[i]	lcp[i]	up[i]	down[i]	nextLIndex[i]	s[sa[i]]
0	10	0		1	2	i$
1	4	1				issippi$
2	0	0	1		3	mississippi$
3	8	0			4	ppi$
4	6	0		5		sippi$
5	2	1				ssissippi$

dobivaju se redom iz $childtab[i_1].nextLIndex$, ..., $childtab[i_{k-1}].nextLIndex$. U [1] je dokazano da su sada child intervali LCP intervala [i..j] redom $[i..i_1 - 1]$, $[i_1..i_2 - 1]$, ..., $[i_k..j]$.

3.2. Konstrukcija child polja

Algoritam 3.1 prikazuje izgradnju rijetkog child polja. Originalni algoritam dokazan je u [1], a autori u [4] modificiraju ga za rad s rijetkim sufiksnim poljem. Kao što je rečeno u odjeklju 2.2, za izgradnju child polja potrebno je imati samo informaciju sadržanu u LCP intervalima. Obzirom da uvođenje parametra rijetkosti K ne utječe na definiciju LCP intervala, moguće je izgraditi rijetko child polje.

U prvom linearnom prolazu LCP polja računaju se `up` i `down` vrijednosti. Trenutni indeks stavlja se na vrh stoga ukoliko je njegova LCP-vrijednost veća ili jednaka LCP-vrijednosti s vrha stoga. U suprotnom, elementi se skidaju sa stoga dok su njihove LCP-vrijednosti veće od LCP-vrijednosti trenutnog indeksa. Usporedbom LCP-vrijednosti trenutnog indeksa i indeksa s vrha stoga, `up` i `down` vrijednostima se pridjeljuju elemenati skinutih sa stoga tijekom prolaska.

U odnosu na [1] uvedena je manja modifikacija u dio algoritma koji računa `up` i `down` vrijednosti kako bi algoritam radio za nizove koji ne sadrže završni znak koji je leksikografski veći od svih ostalih. U suprotnom, `down` vrijednosti posljenjeg rastućeg niza LCP vrijednosti mogu ostati na stogu na kraju algoritma. Kako bi se ovo riješilo, stog se prazni petljom koja provjerava mogu li se naći dodatne `down` vrijednosti.

U drugom linearnom prolazu LCP polja računaju se `nextLIndex` vrijednosti. Izračun `nextLIndex` vrijednosti je jednostavniji. Potrebno je usporediti LCP vrijednost trenutnog indeksa s LCP vrijednošću indeksa s vrha stoga. Ukoliko su navedene vrijednosti jednake, `nextLValue` u child polju na indeksu s vrha stoga postaje trenutni indeks. Ovime završava izgradnja child polja s linearnom vremenskom i memorijskom složenošću.

Algoritam 3.1: Algoritam za konstrukciju child polja

```
1  function constructChildArray(LCP) begin
2      lastIndex = −1;
3      ST.push(0);
4      for i = 1 to n/K − 1 do
5          while LCP[i] < LCP[ST.top] do
6              lastIndex = ST.pop;
7              if LCP[i] ≤ LCP[ST.top] and LCP[ST.top] ≠ LCP[lastIndex]
                then
8                  CHILD[ST.top].down = lastIndex;
9          if lastIndex ≠ −1 then
10             CHILD[i].up = lastIndex;
11             lastIndex = −1;
12         ST.push(i);
13     while 0 < LCP[ST.top] do
14         lastIndex = ST.pop;
15         if 0 ≤ LCP[ST.top] and LCP[ST.top] ≠ LCP[lastIndex] then
16             CHILD[ST.top].down = lastIndex
17     for i = 1 to n/K − 1 do
18         while LCP[i] < LCP[ST.top] do
19             ST.pop;
20         if LCP[i] = LCP[ST.top] then
21             lastIndex = ST.pop;
22             CHILD[lastIndex].nextLIndex = i;
23         ST.push(i);
```

4. Rezultati

U ovom poglavlju predstavljeni su rezultati implementacije child polja. Testovi su izvršeni nad generiranim nizovima te nad stvarnim genomima. Implementacija razvijena u sklopu ovog projekta nalazi se na repozitoriju `https://github.com/arazum/bio`, a referentno rješenje nalazi se ovdje `https://github.com/readmapping/essaMEM`.

U idućem odjeljku opisan je način vrednovanja rješenja te generiranja i dohvaćanja primera. U odjeljku 4.2 prikazana je usporedba vremena izvođenja ovog i referentnog rješenja. U odjelju 4.3 prikazana je usporedba utroška memorije. U odjeljku 4.4 prikazane su ovisnosti vremena izvođenja i utroška memorije o broju baza i parametru K.

4.1. Vrednovanje rješenja

U svrhu usporedivosti, referentno rješenje modificirano je na način da je nakon konstrukcije child niza isti ispisan i izvršavanje programa je prekinuto. Ovako je postignuto da referentno rješenje izvrši isti dio cjelokupnog algoritma kao i rješenje razvijeno u radu kako bi se vrijeme izvršavanja i količina utrošene memorije mogli ispravno usporediti. U rješenju u radu je također na poslijetku ispisan child niz.

Za vrednovanje rješenja napravljen je odvojeni program. Program za svaki test primjer iz zadanog skupa primjera pokreće razvijeno rješenje te referentno rješenje i uspoređuje njihove ispise. U slučaju ne poklapanja, program ispisuje grešku. Uz usporedbu rezultata, program prikazuje vrijeme izvršavanja rješenja i utrošenu memoriju te prikazuje poboljšanje istih u odnosu na referentno rješenje. Za mjerenje vremena izvršavanja i utrošene memorije korišten je program `cgmemtime`.

Rješenja su vrednovana nad primjerima u *FASTA* formatu s brojem baza od 10^5 do 10^7 uz različite vrijednosti parametra K. Za generiranje primjera korišten je jednostavi odvojeni program. Korišteni stvarni primjeri preuzeti su s web lokacije `http://bacteria.ensembl.org/`, a sadrže genome organizama: *Escherichia coli*,

Helicobacter pylori, *Methylobacillus flagellatus*.

4.2. Vrijeme izvođenja

Prilikom traženja MEM-ova potrebno je paziti na vrijeme izvođenja. Kako se MEM-ovi traže nad velikim nizovima, rješenje problema prirodno teži k većem vremenu izvođenja. Potrebno je optimirati sve korake algoritma, tako i izgradnju child polja, kako bi se ostvarilo što kraće izvođenja.

Tablice 4.1 i 4.2 prikazuju vrijeme konstrukcije child polja. Stupac t_{impl} predstavlja vrijeme izvođenja izvedene implementacije dok stupac t_{ref} predstavlja vrijeme izvođenja referentnog rješenja. Stupac $t_{razlika}$ predstavlja razliku u vremenu izvođenja između izvedene implementacije i referentnog rješenja. Vrijednosti stupca $t_{razlika}$ dobiveni su izrazom $100\% \cdot \frac{t_{ref} - t_{impl}}{t_{ref}}$.

Tablica 4.1 prikazuje usporedbu vremena izgradnje child polja nad postojećim genomima. Može se uočiti kako su vremena izvođenja izvedene implementacije i referentnog rješenja vrlo slična.

Tablica 4.1: Usporedba vremena izvršavanja izvedene implementacije i referentnog rješenja za različite primjere i vrijednosti parametra K

Primjer	Broj baza	K	t_{impl}	t_{ref}	$t_{razlika}$
E. coli	5676107	5	0.776514	0.861211	9.83%
E. coli	5676107	20	5.30142	5.3719	1.31%
E. coli	5676107	100	9.01133	8.96336	-0.53%
E. coli	4393089	5	0.579196	0.697147	16.91%
E. coli	4393089	20	4.17962	4.21829	0.91%
E. coli	4393089	100	6.95086	6.83971	-1.62%
Helicobacter	1617653	5	0.168793	0.198376	14.91%
Helicobacter	1617653	20	1.65821	1.59517	-3.95%
Helicobacter	1617653	100	2.62	2.59599	-0.92%
Methylobacillus	2971517	5	0.353354	0.389577	9.29%
Methylobacillus	2971517	20	2.95769	2.95796	0.00%
Methylobacillus	2971517	100	4.71582	4.66719	-1.04%

Tablica 4.2 prikazuje usporedbu vremena izgradnje child polja nad generiranim nizovima. Vrijeme izvođenja izvedene implementacije i referentnog rješenja vrlo je slično te vrijede prethodni zaključci.

Tablica 4.2: Usporedba vremena izvršavanja izvedene implementacije i referentnog rješenja za različite generirane primjere i različite vrijednosti parametra K

Broj baza	K	t_{impl}	t_{ref}	$t_{razlika}$
10^5	5	0.007086	0.014893	52.42%
10^5	20	0.124813	0.127244	1.91%
10^5	100	0.181825	0.15902	-14.34%
10^6	5	0.106828	0.141809	24.66%
10^6	20	1.05831	1.04681	-1.09%
10^6	100	1.59274	1.59819	0.034%
10^7	5	1.41559	1.56725	9.67%
10^7	20	9.81479	9.53324	-2.95%
10^7	100	15.609	15.5091	-0.064%

4.3. Potrošnja memorije

Traženje MEM-ova nad genomima može dovesti do značajnih otkrića. Kako je dostupan sve veći broj cijelovitih genoma, algoritmi rade nad vrlo velikim nizovima. Zbog toga je potrebno voditi računa o efikasnosti korištenja memorije.

Child polje je jedna od ključnih struktura podataka algoritma essaMEM i potrebno ju je konstruirati na efikasan način, odnosno da se ne troši više memorije no što je to zaista potrebno.

Tablice 4.3 i 4.4 prikazuju utrošak memorije child polja. Stupac mem_{impl} predstavlja utrošak memorije izvedene implementacije dok stupac mem_{ref} predstavlja utrošak memorije referentnog rješenja. Stupac $mem_{razlika}$ predstavlja razliku utrošene memorije između izvedene implementacije i referentnog rješenja. Vrijednosti stupca $mem_{razlika}$ dobiveni su izrazom $100\% \cdot \frac{mem_{ref} - mem_{impl}}{mem_{ref}}$.

Iz rezultata prikazanih tablicama 4.3 i 4.4 može se vidjeti kako izvedena implementacija ima manji utrošak memorije od referentnog rješenja za gotovo sve primjere.

Tablica 4.3: Usporedba utrošene memorije izvedene implementacije i referentnog rješenja za različite primjere i vrijednosti parametra K

Primjer	Broj baza	K	mem_{impl}	mem_{ref}	$mem_{razlika}$
E. coli	5676107	5	24988	29652	15.72%
E. coli	5676107	20	11824	14004	15.56%
E. coli	5676107	100	9772	12908	24.29%
E. coli	4393089	5	21292	25108	15.19%
E. coli	4393089	20	11464	12016	4.59%
E. coli	4393089	100	11000	10180	-8.05%
Helicobacter	1617653	5	7976	15416	48.26%
Helicobacter	1617653	20	4156	4752	12.54%
Helicobacter	1617653	100	3376	4760	29.07%
Methylobacillus	2971517	5	13752	20012	31.28%
Methylobacillus	2971517	20	6552	7312	10.39%
Methylobacillus	2971517	100	6456	8172	20.99%

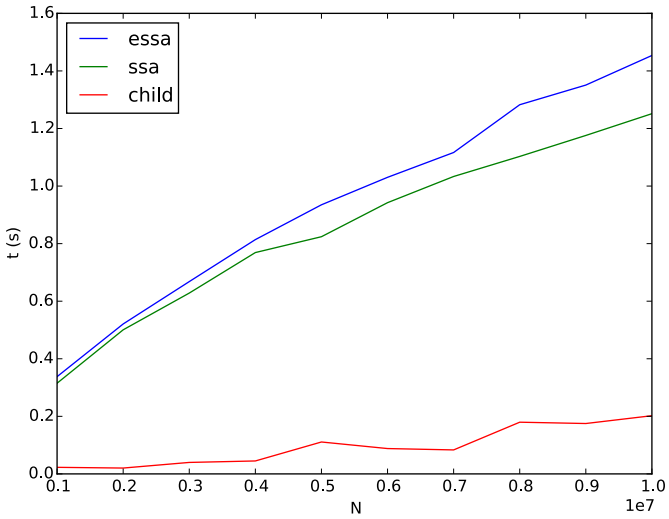
Tablica 4.4: Usporedba utrošene memorije izvedene implementacije i referentnog rješenja za različite generirane primjere i različite vrijednosti parametra K

Broj baza	K	mem_{impl}	mem_{ref}	$mem_{razlika}$
10^5	5	1880	10160	81.49%
10^5	20	1664	1832	9.17%
10^5	100	1736	1848	6.06%
10^6	5	5288	13208	59.96%
10^6	20	3100	3432	9.67%
10^6	100	2568	3436	25.26%
10^7	5	43268	44880	3.59%
10^7	20	20780	22264	6.66%
10^7	100	19076	21444	11.04%

4.4. Ovisnost utroška resursa o parametrima

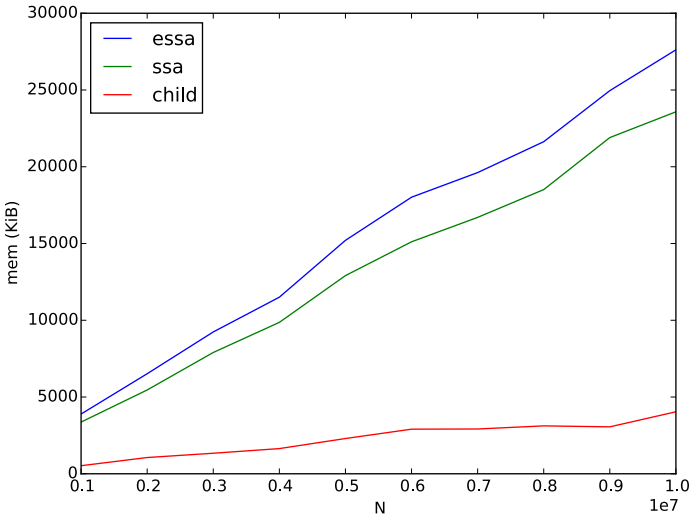
U nastavku su prikazani grafovi ovisnosti utroška resursa o broju baza N i o parametru rijetkosti K. Na svakom grafu prikazane su tri krivulje: *essa* – utrošak cjelokupnog rješenja,, *ssa* – utrošak dijela rješenja do konstrukcije child polja te *child* – utrošak konsturkcije child polja. Kako se *ssa* dio rješenja nalazi na početku, u izvršavanju ga je moguće izdvojiti i vršiti mjerenja samo za njega. *child* dio se onda može dobiti oduzimanjem *ssa* dijela od *essa*. Na ovaj način dobivamo približan utrošak resursa potrebnih za konstrukciju child niza.

Vrijeme izvršavanja i utrošena memorija su u linearnoj ovisnosti o broju baza što se vidi na slikama 4.1 i 4.2. Ovaj rezultat je očekivan prema algoritmu.

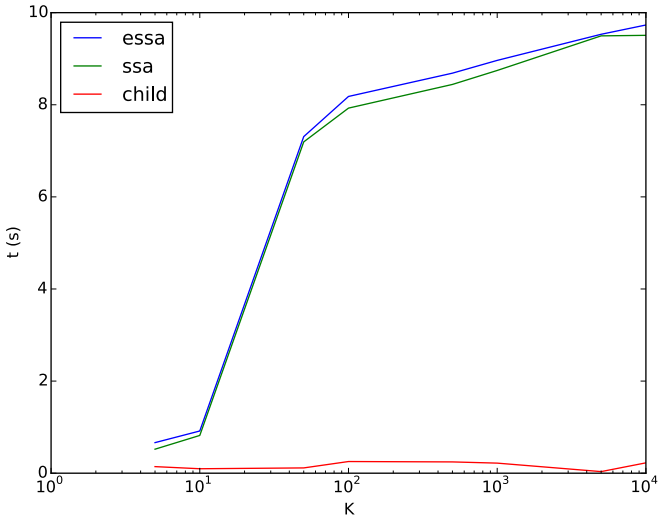


Slika 4.1: Ovisnost vremena izvršavanja o broju baza ($K = 10$)

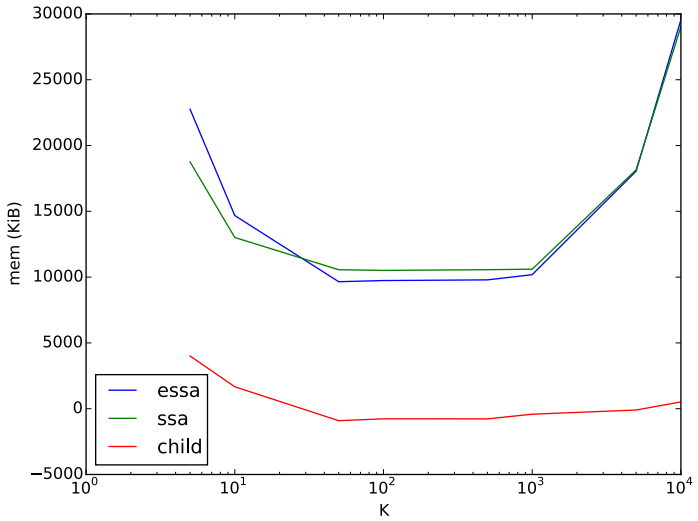
Na slici 4.3 vrijednost *child* krivulje kreće se oko nule. To je zato jer je za veće vrijednosti parametra K dominantan po utrošku resursa dio algoritma koji konstruira sufiksno polje. Na slici 4.3 može se vidjeti ovisnost utroška memorije obrnuto proporcionalna parametru K, što je i očekivano.



Slika 4.2: Ovisnost utrošene memorije o broju baza ($K = 10$)



Slika 4.3: Ovisnost vremena izvršavanja o parametru K ($N = 5 \cdot 10^6$)



Slika 4.4: Ovisnost utrošene memorije o parametru K ($N = 5 \cdot 10^6$)

5. Zaključak

Pronalaženje maksimalnih točnih slaganja je vrlo bitan alat u komparativnoj genomici jer omogućuje izračunavanje i rangiranje sličnosti genoma te utvrđivanje ključnih točaka prilikom uspoređivanja genoma. Time se mogu pronaći sličnosti i razlike između organizama.

Sufiksna stabla su prva struktura koja je omogućila efikasno uspoređivanje nizova. Razvojem područja razvila su se sufiksna polja. Ona omogućuju stvaranje virtualnih sufiksnih stabala te donose poboljšanja u iskoristivosti resursa. Cijena toga je složenost izgradnje takve strukture.

Napredak u razvoju sufiksnih polja omogućuje izgradnju efikasnijih algoritama za pronalazak najvećih točnih slaganja. essaMEM predstavlja poboljšanje dosadašnjih metoda. To je složen postupak pogodan za pronalaženje MEM-ova nad velikim nizovima.

U sklopu ovog projektnog zadatka implementiran je algoritam izgradnje child polja, jedne od ključnih struktura algoritma essaMEM. Implementirana inačica postiže vrlo slične rezultate kao i referentno rješenje. Vremensko izvođenje implementiranog algoritma izgradnje child polja je u većini slučajeva kraće od referentnog rješenja te je memorijski utrošak manji.

6. Literatura

[1] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, i Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.

[2] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, i Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. U *Annual Symposium on Combinatorial Pattern Matching*, stranice 181–192. Springer, 2001.

[3] Zia Khan, Joshua S Bloom, Leonid Kruglyak, i Mona Singh. A practical algorithm for finding maximal exact matches in large sequence datasets using sparse suffix arrays. *Bioinformatics*, 25(13):1609–1616, 2009.

[4] Michaël Vyverman, Bernard De Baets, Veerle Fack, i Peter Dawyndt. essamem: finding maximal exact matches using enhanced sparse suffix arrays. *Bioinformatics*, 29(6):802–804, 2013.

essaMEM: finding maximal exact matches using enhanced sparse suffix arrays

Sažetak

Komparativna genomika je područje bioinformatike u kojem se značajke genoma različitih organizama uspoređuju. Rastom broja dostupnih potpunih genoma organizama javlja se zahtjev za novim unaprijeđenim tehnikama usporedbe genoma. Bitan alat u komparativnoj genomici je traženje maksimalnih točnih slaganja. essaMEM je napredan algoritam za efikasno računanje maksimalnih točnih slaganja za velike nizove. Cilj ovog rada bio je proučiti algoritam essaMEM te implementirati izgradnju child polja. Na kraju rada predstavljeni su rezultati izgradnje child polja te su uspoređeni s referentnim rješenjem.

Ključne riječi: essaMEM, MEM, enhanced sparse suffix array, child array