Bioinformatika

Traženje podudarnih nizova uz k različitih znakova

FER, 2016/2017

David Romić; 0036465852

Vilim Stubičan; 0036475140

Domagoj Korman; 0036474986

Problem

Podudaranje teksta, odnosno problem pronalaženja svih podudaranja nekog uzorka u tekstu je klasičan problem u računalnoj znanosti. Razvojem znanosti ljudi su shvatili kako se taj isti problem može primjeniti i na različite druge znanosti kao što je bioinformatika. Zbog toga se podudaranje teksta podijelilo na egzaktna podudaranja gdje tražimo točno određeni uzorak u zadanom tekstu i približna podudaranja u kojima tražimo vrlo slične uzorke u zadanom tekstu.

Prbiližna podudaranja se definiraju tako da se na neki uzorak definira broj mogućih pogrešaka. Zatim algoritam u zadanom tekstu pronađe sva podudaranja koja se podudaraju s do broja mogućih pogrešaka.

Glavni problem kod takvog pretraživanja teksta je složenost programa koji pretražuje zato što algoritam istovremeno treba uspoređivati uzorak, tekst i pamtiti koliko je pogrešaka dozvoljeno. Najjednostavniji način pretraživanja je naivnim algoritmom čija je složenost O(n*m).

Cilj ovog projekta je bio napraviti algoritam koji će uz predobrađivanje moći pretražiti tekst uz manju složenost od naivnog algoritma.

Algoritmi

Bazni algoritam

Implementacija algoritma za pretraživanje se temelji na istraživanju "*On string matching with k mismatches" (M. Nicolae,S. Rajasekaran)* u kojem proučava više različitih pristupa uspoređivanju uzoraka unutar dva teksta. Jedan od navedenih se sastoji od provjeravanja podskupa uzoraka iz cjelokupnog teksta koji se promatra. Povučeni algoritmom broj šest iz navedenog rada, implementirali smo svoju verziju istog algoritma. Slijedi pseudo kod originalnog navedenog algoritma.

```
// S = podskup pozicija u tekstu koje se provjeravaju
// M = lista s podacima o razlikama u uzorku i tekstu na poziciji
// n = duljina teksta; m = duljina uzorka;
// T = tekst; P = uzorak
za a ∈ S radi M[a] = 0;
i=1;
dok je i ≤ n radi
      // Pronađi najveći l, takav da je T_{i..i+l-1} = P_{j..j+l-1} za neki j;
      za a ∈ S gdje vrijedi a ≤ i < a + m radi
      M[a] = updateMismatches(M[a], i − a + 1, j, l);
      ako je M[a] > k onda S = S − {a};
      i=i+l+1;
vrati {a ∈ S|M[a] ≤ k}
```

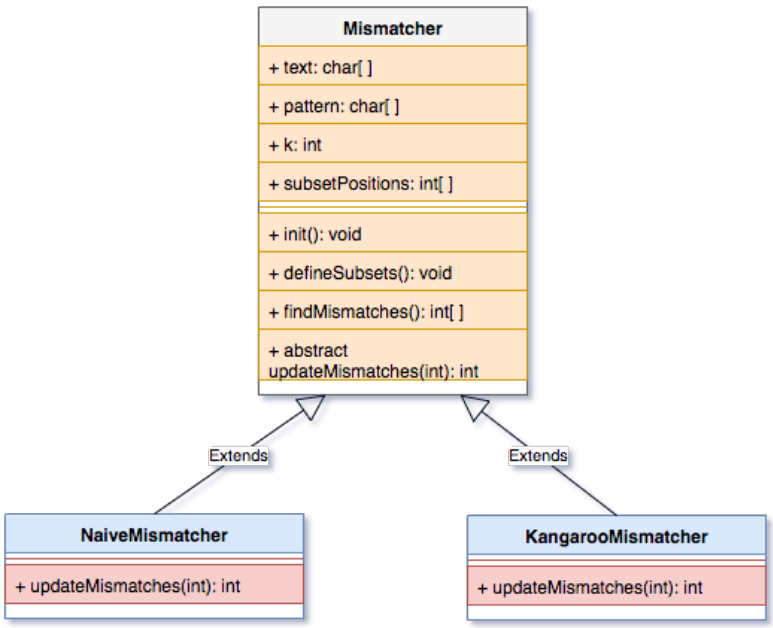
Algoritam se temelji na filtraciji stanja koja bi mogla biti u uzorku. Ovo se odradi u koraku gdje se pronalazi najveći ***l*** za koji vrijedi $T_{i..i+l-1} = P_{j..j+l-1}$. Za određenu poziciju ***i*** u tekstu provjeravamo da li se nalazi igdje u uzorku. Ukoliko se pronađe pozicija koja ima određenu duljinu, slijedi provjera stanja. Stanja koja se provjeravaju su ona čiji prozor duljine uzorka sadržava poziciju ***i*** te ćemo ih nazvati "kandidat" stanjima. "Kandidat" stanja ulaze u provjeru razlika te osvježavaju vrijednost unutar liste s podacima o razlikama. Ukoliko je pronađeno više od dozvoljenih razlika, "kandidat" stanje se uklanja iz liste mogućih stanja koja se podudaraju. Algoritmi koji se koriste unutar *updateMismatches* metode su opisani u sljedećim dijelovima.

Sada predstavljamo našu inačicu istog algoritma.

```
// S = podskup pozicija u tekstu koje se provjeravaju
// M = lista s podacima o razlikama u uzorku i tekstu na poziciji
// n = duljina teksta; m = duljina uzorka;
// T = tekst; P = uzorak
za a ∈ S radi M[a] = -1;
i=1;
dok je i ≤ n - m + 1 radi
      // Pronađi najveći l, takav da je T_{i..i+l-1} = P_{j..j+l-1} za neki j;
      za a ∈ S gdje vrijedi a ≤ i < a + m radi
      M[a] = updateMismatches(a);
      ako je M[a] > k onda S = S − {a};
      ako je l > 0 onda
            i = i + l;
      Inače
            i = i + 1;
vrati {a ∈ S|M[a] ≤ k & M[a] > -1}
```

Glavna razlika se sastoji u proširenju početnih stanja s oznakom provjere stanja. Ukoliko je konačna vrijednost u listi s podacima o razlikama jednaka -1, tada navedeno stanje nije provjereno. Dodatno, u metodu provjere se sada prosljeđuje pozicija koja se treba provjeriti. Metode kojima se pozicije provjeravaju su opisane u sljedećem dijelu.

Konkretna implementacija se vidi na sljedećem dijagramu razreda.



Bazni razred *Mismatcher* se brine oko obavljanja temeljnog dijela algoritma, a nasljeđeni razredi *NaiveMismatcher* i *KangarooMismatcher* implementiraju različite metode provjere razlike uzoraka.

Naivni algoritam

Naivni algoritam je najjednostavnija inačica provjere dva teksta. Temelji se na dvostrukom prolazu originalnog teksta i danog uzorka. Svaki element se obrađuje te se algoritam vrti u vremenskoj složenosti od *O(m)* nad jednom pozicijom gdje je *m* duljina uzorka, što u konačnici rezultira vremenskom složenosti od *O(n*m)* za svaku od traženih *n* pozicija u tekstu. Slijedi pseudokod metode *updateMismatches* implementiran naivnim algoritmom.

```
updateMismatches(int position)
      c = 0;
      za a = 0 do m
            ako je T_position+a != P_a onda c = c + 1;
      vrati c;
```

Ovisno o implementaciji spremanja teksta i uzorka, prostorna složenost algoritma može biti *O(n + m)* jer zahtjeva samo spremanje teksta i uzorka. Glavna mana algoritma je prolaženje cijelog uzorka i pozicije unutar teksta za svaku “kandidat” poziciju.

Kangaroo algoritam s predobrađivanjem

Kangaroo algoritam s predobrađivanjem bi trebao biti brži od naivnog pretraživanja. Glavna ideja algoritma je da ubrza način pretrage jednakih podskupova između uzorka i teksta.

Glavne sastavnice algoritma su:

- Sufiksno polje teksta i uzorka
- *Longest common prefix (LCP)*
- *Range minimum query (RMQ)*
- *Kangaroo search*

Sufiksno polje

To je polje sufiksa koje se generira od teksta i uzorka koji se pretražuje. Sufiksno polje se koristi samo za predobrađivanje te se ne koristi u samom algoritmu pretraživanja. Jako je bitno sortirati sufikse abecedno nakon što se polje izgenerira jer je potrebno za generiranje LCP-a.

Longest common prefix

LCP je polje u koje spremamo dužinu jednakih prefixa za trenutnu vrijednost sufiksnog polja i sljedeću vrijednost sufiksnog polja.

```
Lcp[i] = prefixLength(suffix[i], suffix[i+1])
```

Ideja LCP-a je kako ne bi morali provjeravati znak po znak već možemo odmah iz LCP-a vidjeti poklapaju li se tekst i uzorak.

Kako bi LCP imao smisla, sufiksno polje mora biti abecedno sortirano.

Range minimum query

RMQ je algoritam koji se koristi kako bi efektivno pronašli dužinu jednakih prefixa između bilo koja dva uzorka koja se već nalaze u postojećem LCP polju.

Postoji mnogo implementacija RMQ-a te brzina samog algoritma uvelike ovisi o tome kako je RMQ implementiran. RMQ generalno radi tako da pronađe indekse oba uzorka u LCP polju te vrati minimalnu vrijednost LCP-a između ta dva indeksa.

Kangaroo search

To je algoritam pretraživanja teksta koji bi uz korištenje ranije objašnjenih struktura trebao biti brži od naivnog algoritma jer radi na temelju preskakanja znakova za koje možemo ustvrditi da su sigurno jednaki u oba uzorka.

Smisao algoritma je da uz korištenje RMQ-a dohvati duljinu prefixa koji je jednak u oba uzorka, tu duljinu ćemo nazvati `L`. Ono što možemo zaključiti je da se na `L + 1` mjestu nalazi različit znak.

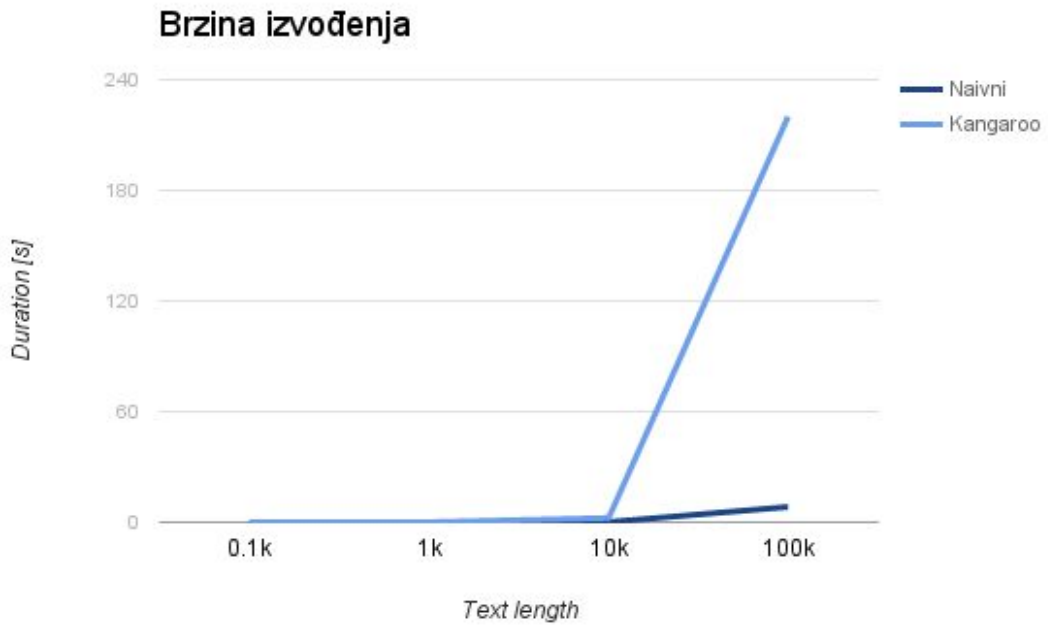
Nakon toga, preskočimo na mjesto `L + 2` u oba uzorka te ih nastavimo uspoređivati korištenjem RMQ-a. Ako u nekom trenutku broj različitih znakova bude veći od dopuštenog, možemo zaustaviti pretraživanje uzoraka, zaključiti da ti uzorci nisu podudarni te prelazimo na sljedeći uzorak.

Uzorci su podudarni ako pronađemo dva uzorak u kojemu `L` dođe do kraja uzorka, a prije toga ne prekinemo algoritam.

Vidljivo je kako bi algoritam trebao biti brži zato što nam omogućuje preskakanje podudarnih znakova i odmah pronalazi mjesto gdje se nalazi prvi različiti znak.

Rezultati

Brzina



Brzina izvođenja naivnog algoritma je na kraju ispala brža od brzine izvođenja *kangaroo* algoritma. Glavni razlog toga je složenost RMQ-a koji je glavna mana samog algoritma.

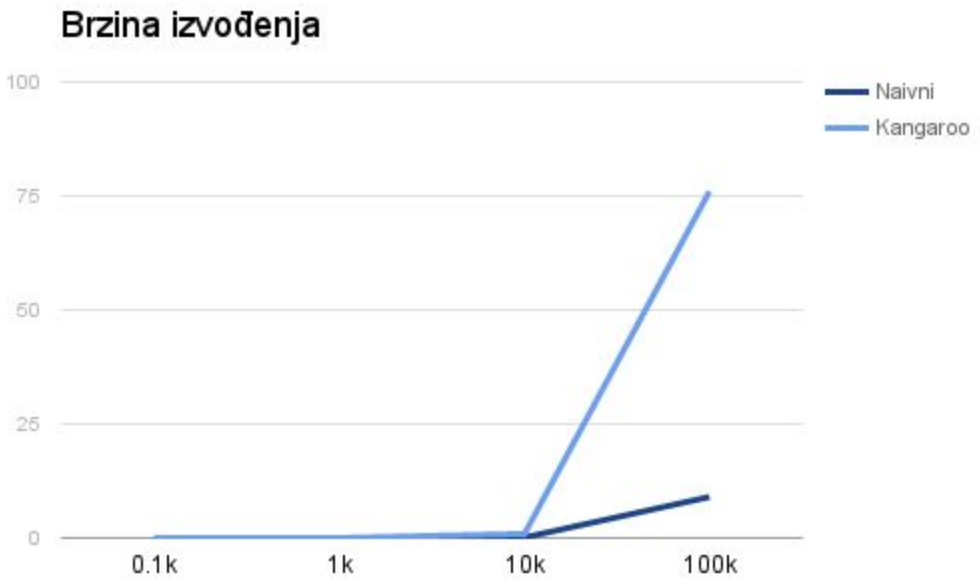
Testirali smo *Kangaroo* algoritam gdje smo RMQ implementirali na različite načine od najjednostavnijeg, do spremanja indexa u hash mapu pa sve do matrice u kojoj smo imali spremljene vrijednosti RMQ-a za svaka dva sufixa teksta i patterna. Najbrži način je bio zadnji, no nažalost, vrijeme predobradbe je tada bilo presporo pa smo ostavili RMQ da radi na najjednostavniji način kako bi se algoritam izvršio u nekom normalnom vremenu.

Također smo primjetili kako algoritam jako uspori stvaranje novih objekata prilikom usporedbe uzoraka što se vjerojatno može izbjeći direktnim pristupanjem vrijednostima preko memorije.

Isto tako, pozivanje metode koja vrati vrijednost uspori rad algoritma, a imamo dosta metoda koje vraćaju neke vrijednosti. Sam po sebi, *kangaroo* algoritam ima puno više provjera što isto utječe na brzinu algoritma.

Kada bi ispravili sve navedene probleme, *kangaroo* algoritam bi vjerojatno bio brži od naivnog za veće uzorke.

Brzina v2



Nakon što smo izmjenili i ispravili navedene probleme algoritam radi puno brže.

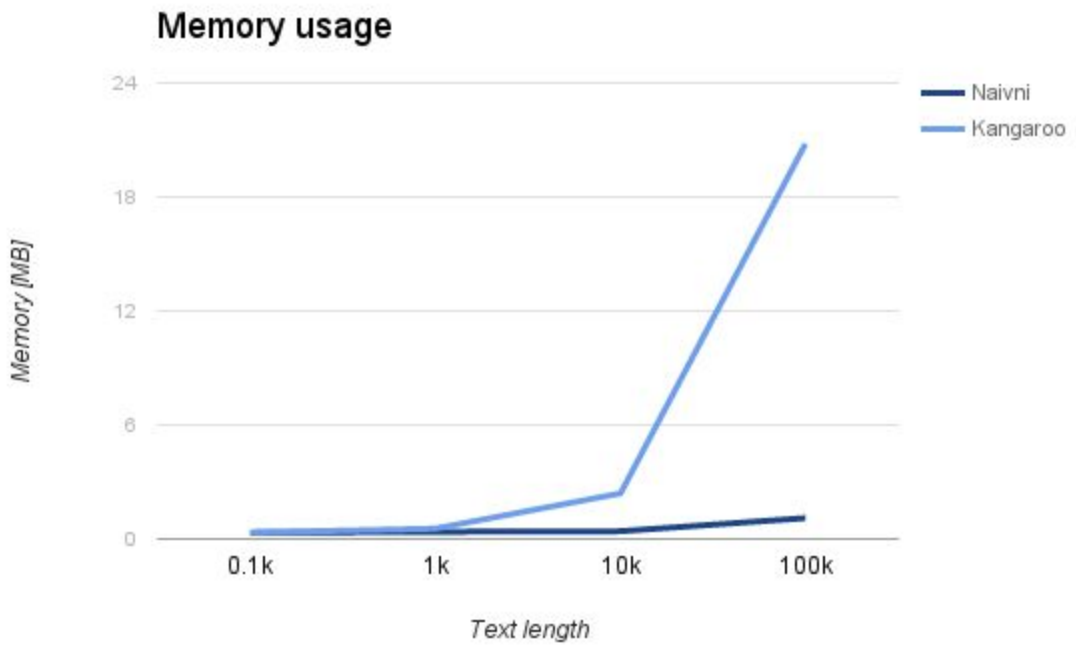
Izmjene koje smo napravili kako bi ubrzali algoritam su:

- Brži način stvaranja LCP polja
- Pristup LCP vrijednostima koristeći inverzno sufiksno polje
- Nema stvaranja novih objekata jer se sve radi preko pokazivača na tekst i indeksa

Sufiksno polje se generira slanjem inputa kao tekst#uzorak. Iz takvog sufiksnog polja možemo jednostavno usprediti uzorak i tekst.

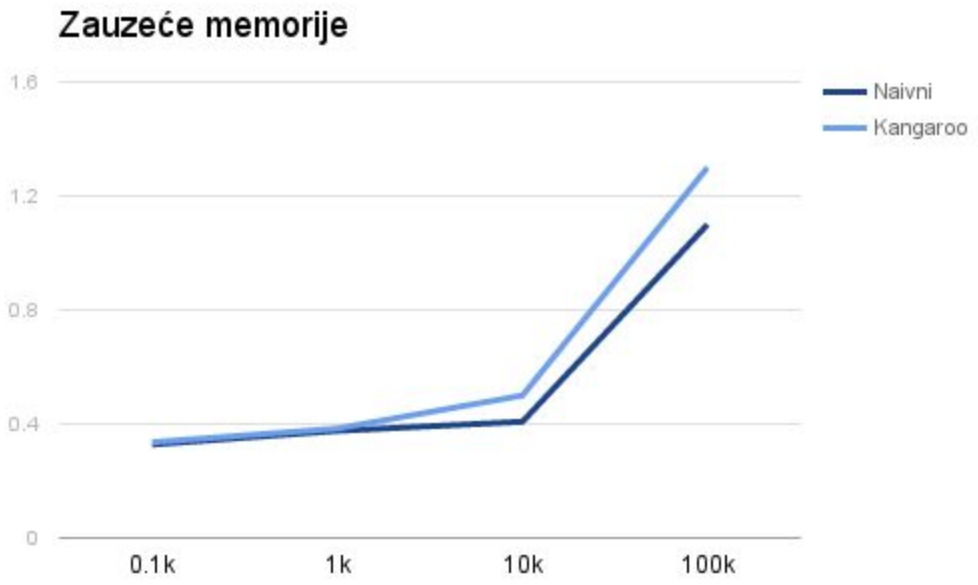
Zbog načina pristupa LCP vrijednostima korištenjem inverzno sufiksno polje, smanjili smo složenost na O(1).

Memorija



Logično je i očito da *kangaroo* algoritam zauzima više memorije od naivnog algoritma. *Kangaroo* algoritam ima cijeli proces predobradbe cijelog teksta, gdje sprema stvoreno sufiksno polje i LCP vrijednosti za sve sufikse. Za razliku od njega, naivni algoritam nema potrebe za dodatnom memorijom. On u memoriji ima spremljene uzorke koje uspoređuje te ne stvara nikakva dodatna polja.

Memorija v2



U novom algoritmu se samo stvaraju nova polja za sufiksno polje i inverzno sufiksno polje. Ostalo se sve izvršava korištenjem indexa pa nije potrebno mnogo više memorije od naivnog algoritma.

Zaključak

Promatrajući algoritam naveden u ovom radu možemo vidjeti veliku prednost nad jednostavnim implementacijama poput naivnog algoritma nad cijelim tekstom. Dodatno, uočavamo veliku razliku u brzini prilikom rada na manjem podskupu uzoraka. Razmjena koja se obavi za navedenu brzinu je apsolutna točnost koja se može opravdati statističkom razdiobom rezultata u velikim skupovima.

Naše provjere su dovele do obrnutih očekivanja između dva algoritma koja se koriste za provjeru razlika između dva uzorka. Ova neočekivana razlika se jasno očituje iz same jačine glavnog algoritma koji omogućava efikasnu filtraciju poduzoraka stanja do takve razine da interne implementacije naivnog algoritma i kangaroo algoritma utječu na samu brzinu. Složenost kangaroo algoritma zahtjeva dodatno pozivanje internih metoda koje se postavljaju na stog i miču veliki broj puta prilikom provjere što rezultira sporijom provjerom nad uzorcima gdje ima puno razlika.

Kao zaključak slijedi da je algoritam opisan u ovom radu vrlo efikasan neovisno o podalgoritmu za provjeru samih razlika unutar odabranih poduzoraka.

Literatura

1. “Fast k mismatch string matching algorithm”, *http://cs.stackexchange.com/questions/4797/fast-k-mismatch-string-matching-algorithm/4855#4855*, stranica posjećena: 15. Siječnja 2017.
2. “Range Minimum Query and Lowest Common Ancestor”, *https://www.topcoder.com/community/data-science/data-science-tutorials/range-minimum-query-and-lowest-common-ancestor/*, stranica posjećena: 15. Siječnja 2017.
3. Nicolae M., Rajasekaran S. “On string matching with k mismatches”, *https://arxiv.org/pdf/1307.1406.pdf*, stranica posjećena: 15. Siječnja 2017.
4. “String Matching with k Mismatches by Using Kangaroo Method”, *http://t2.ecp168.net/webs@73/cyberhood/Approximate_String_Matching/String_Matching_with_k_Mismatches_by_Using_Kangaroo_Method.ppt*, stranica posjećena: 15. Siječnja 2017.
5. *https://github.com/mariusmni/kmismatches*, stranica posjećena: 15. Siječnja 2017.
6. “kasai’s Algorithm for Construction of LCP array from Suffix Array”, *http://www.geeksforgeeks.org/%C2%AD%C2%ADkasais-algorithm-for-construction-of-lcp-array-from-suffix-array/*, stranica posjećena: 15. Siječnja 2017.
7. *https://github.com/hermanstehouwer/Enhanced-Suffix-Arrays*, stranica posjećena: 15. Siječnja 2017.