SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

IZVJEŠTAJ

Izgradnja sufiksnog polja korištenjem SACA-K algoritma

Matija Folnović, Paula Gombar

Predmet: *Bioinformatika*

Voditeljica: *Mirjana Domazet-Lošo*

Zagreb, siječanj 2017.

Sadržaj

1. Uvod

U sklopu ovog projekta, potrebno je implementirati izgradnju sufiksnog polja korištenjem SACA-K algoritma. Za početak je bitno shvatiti što je sufiksno polje, zašto se koristi, koje su postojeće implementacije i razlike između njih te koje novine donosi SACA-K algoritam.

Sufiksno polje je struktura podataka koja omogućava brz pronalazak kratkih podnizova u zadanom slijedu ili sljedovima. Drugim riječima, koristeći tu strukturu možemo brzo indeksirati bilo kakav dugačak tekst, što je posebno bitno u području bioinformatike, jer je čest problem pronalaženja pojavljivanja uzorka u nekom zadanom tekstu, npr. genomu. Korištenjem sufiksnog stabla moguće je riješiti i mnoštvo drugih složenih operacija nad znakovnim nizovima, poput pronalaska najduljeg zajedničkog podniza dvaju nizova u vremenu proporcionalnom zbroju njihovih duljina (Gusfield, 1997).

Sufiksno polje u biti predstavlja niz početnih pozicija sufiksa u tekstu, sortiranih leksikografskim redom. Definiramo S kao niz znakova duljine n čiji su znakovi elementi abecede Σ. Također, neka niz S završava posebnim znakom $ koji je leksikografski manji od svih ostalih znakova abecede Σ i ne pojavljuje se nigdje drugdje unutar S. Definirajmo S[i, j] kao podniz niza S koji započinje na i-tom, a završava na j-tom indeksu. Sufiks niza S je svaki njegov podniz koji završava posljednjim znakom, tj. S[i, n], $0 \leq i < n$. Sufiks koji počinje na i-tom indeksu označavat ćemo sa suf(S, i). Sufiksno polje (engl. suffix array) SA je niz cijelih brojeva koji označavaju početne pozicije abecedno poredanih sufiksa niza S. Dakle, vrijedi da SA[i] označava početnu poziciju i-tog najmanjeg sufiksa niza S.

Pogledajmo na primjeru kada je S = “banana$”:

i	**1**	**2**	**3**	**4**	**5**	**6**	**7**
S[i]	b	a	n	a	n	a	$
Suffix	b a n a n a $	a n a n a $	n a n a $	a n a $	n a $	a $	$
SA[i]	7	6	4	2	1	5	3

Polje SA predstavlja sufiksno polje, a tumači se na sljedeći način: SA[3] = 4, što predstavlja sufiks niza S koji počinje na 4. mjestu, tj. “ana$”.

Sufiksno polje može se izgraditi pre-order obilaskom sufiksnog stabla. U stablu korijen predstavlja početni čvor, $, te se dalje grana s obzirom na moguće nastavke (sufikse). Naivna implementacija sufiksnog stabla je pokazana u radu (Ukkonen, 1995) i složenosti je O(n^3), gdje je n duljina ulaznog niza, no već s nekoliko unaprijeđenih pravila dolazi se do gradnje stabla u složenosti od O(n). No, i dalje ostaje problem memorijske složenosti kada radimo sa sufiksnim stablom. Naime, stablo zauzima 20n bajtova memorije, dok sufiksno polje zauzima samo 4n bajtova te je prvi put objašnjeno u radu (Manber & Myers, 1990). Sufiksno polje može zamijeniti sufiksno stablo, odnosno, svaki problem koji se može riješiti korištenjem sufiksnih stabala, može se riješiti i korištenjem sufiksnog polja s istom asimptotskom složenošću, što je pokazano u radu (Abouelhoda et al., 2004).

Postoji mnogo algoritama koji omogućuju konstrukciju sufiksnog polja (engl. Suffix Array Construction Algorithm; SACA). Originalni algoritam koji su osmislili Manber i Myers (1990) omogućavao je izgradnju sufiksnog polja u vremenu O(n log n), za ulazni niz od n znakova. Danas najpopularniji i najbrži algoritam za izgradnju sufiksnog polja u linearnom vremenu je Nong-Zhang-Chanov SA-IS algoritam (Nong et al., 2009, 2011) gdje je vremenska složenost određivanja SA O(n). No, 2013. došlo je do još jedne prekretnice u svijetu konstrukcije sufiksnih polja, kada je u radu (Nong, 2013) opisan poboljšan SA-IS algoritam naziva SACA-K. Eksperimenti su pokazali da je SACA-K 33% brži od SA-IS i koristi manji radni prostor (engl. workspace) za dobivanje sufiksnog polja. Točnije, osim ulaznog stringa i izlaznog SA, koristi još samo radni prostor od K riječi (gdje je K veličina abecede).

2. Opis algoritma SACA-K

Za potpuno shvaćanje algoritma SACA-K, korisno je prvo opisati korake algoritma SA-IS na kojem se SACA-K temelji te istaknuti suptilne razlike između njih. Za početak, potrebno je definirati često korištene pojmove:

- ulazni niz je S i završava s znakom $ koji je abecedno manji od svih znakova abecede Σ
- sufiks koji počinje na mjestu *i* označava se kao s_i = S[i, |S|]
- s_i je S-tip sufiksa (engl. *S-type*) ako je $s_i < s_{i+1}$ ili ako je s_i = $
- s_i je L-tip sufiksa (engl. *S-type*) ako je $s_i > s_{i+1}$
- S[i] je S-tip znaka ako je S[i] < S[i + 1] ili ako je S[i] = S[i + 1] i s_{i+1} je S-tip
- S[i] je L-tip znaka ako je S[i] > S[i + 1] ili ako je S[i] = S[i + 1] i s_{i+1} je L-tip
- S[i] je LMS-znak (engl. leftmost S-type) ako je S[i] S-tip i S[i – 1] L-tip (za i ≥ 1)
- s_i je LMS-sufiks ako je S[i] LMS-znak
- S[i, j] je LMS-podniz ako je S[i, j] znak za kraj niza (i = j) ili ako su S[i] i S[j] LMS-znakovi (i ≠ j), a između njih nema drugih LMS-znakova

Pogledajmo to na primjeru kada je S = "ococonut$". Prolazeći nizom zdesna na lijevo, dobivamo:

S	o	c	o	c	o	n	u	t	$
tip znaka	L	S	L	S	L	S	L	L	S
tip sufiksa	L	S	L	S	L	S	L	L	S
LMS-podnizovi	coc			con		nut$			$

Prvi korak svakog SACA algoritma je upravo određivanje LMS-podnizova. Osnovna ideja SA-IS algoritma, koja omogućuje konstrukciju SA u linearnom vremenu, je takozvano inducirano sortiranje LMS-podnizova. Inducirano sortiranje temeljito je opisano u radu (Nong et al., 2011), a na isti način se provodi u algoritmu SACA-K na dubini 0. Nakon sortiranja LMS-podnizova, svakom se podnizu dodjeljuje novo ime te se tako dobiva novi, skraćeni niz S1, a zatim se rekurzivno računa SA od skraćenog stringa. Pomoćno polje P1 sadrži indekse svih LMS-podnizova, polje B je polje pokazivača na početak ili kraj pojedinog pretinca (sufiksa koji počinju istim početnim znakom iz abecede Σ), dok polje t sadrži zapis tipova znakova u nizu.

U nastavku je prikazan pregled algoritma SA-IS:

SA-IS algoritam (S -> SA):
Ulaz: niz S (|S| = n); S je sastavljen iz znakova iz abecede Σ i završava sa znakom $.
Izlaz: sufiksno polje SA.
Pomoćna polja: t, P1, B

1. Za ulazni niz S odrediti pomoćna polja t (polje tipova znakova: S-tip/L-tip) i polje P1 (polje početnih pozicija LMS-podnizova). |t| = n, |P1| = n1 (n1 je broj LMS-podnizova od S).
2. Inducirano sortirati LMS-podnizove korištenjem P1 i B.
3. Imenovati svaki LMS-podniz prema pripadajućem rangu, čime se dobije novo polje S1.
4. Ako svaki znak iz S1 ima jedinstveno ime, onda se izravno određuje SA1 za S1, a inače pozvati SA-IS(S1, SA1).
5. Odrediti SA iz SA1.

Bitna razlika između algoritama SA-IS i SACA-K je ta što SA-IS koristi pomoćno polje t, P1 i B, dok SACA-K koristi pomoćno polje B samo na dubini 0, dok je zbog drukčijeg načina imenovanja LMS-podnizova u potpunosti izbjegnuta potreba za pomoćnim poljima t i P1.

U nastavku je prikazan pregled algoritma SACA-K:

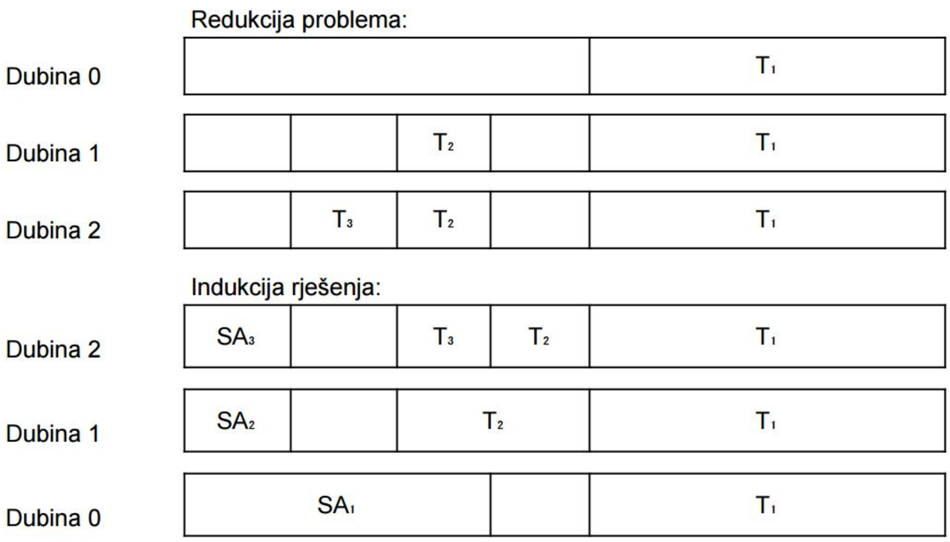
SACA-K algoritam (S -> SA):
Ulaz: niz S (|S| = n); S je sastavljen iz znakova iz abecede Σ i završava sa znakom $.
Izlaz: sufiksno polje SA.
Pomoćna polja: B (samo na dubini 0)
Pomoćne varijable: dubina, K (duljina abecede Σ), n (duljina niza S)

1. faza: inducirano sortirati LMS-podnizova iz S.
    a. Ako je dubina = 0, inducirano sortirati LMS-podnizove koristeći pomoćno polje B (isto kao u SA-IS).
    b. Inače, inducirano sortirati LMS-podnizove koristeći početak ili kraj svakog pretinca kao pokazivač trenutnog pretinca.

2. faza: imenovati svaki LMS-podniz, čime se dobije novo polje S1 (|S1| = n1).
3. faza: rekurzivno sortirati.
    a. Ako je K = n1 (S1 se sastoji od jedinstvenih znakova), izravno odrediti SA1 za S1.
    b. Inače, pozvati SACA-K(S1, SA1, dubina+1).
4. faza: inducirano sortirati SA iz SA1.
    a. Ako je dubina = 0, inducirano sortirati SA iz SA1 koristeći pomoćno polje B.
    b. Inače, inducirano sortirati SA iz SA1 koristeći početak ili kraj svakog pretinca kao pokazivač trenutnog pretinca.

Dvije su novine koje SACA-K uvodi pred drugim SACA algoritmima:

1. SACA-K koristi drukčiji način imenovanja LMS-podnizova, o čemu će biti riječi kasnije.
2. SACA-K koristi prostor polja SA_0 (polje SA na dubini 0) za spremanje $S_1 ... S_n$ (faza redukcije problema) i za spremanje $SA_n ... SA_1$ (faza indukcije rješenja). Prikazano na slici 2.1.



Slika 2.1. Prikaz korištenja polja SA_0.

U nastavku slijede objašnjenja ključnih koraka SACA-K algoritma, popraćena programskim kodom.

2.1 Inducirano sortiranje

Ideja induciranog sortiranja je ono što izdvaja algoritme SA-IS i SACA-K od ostalih SACA algoritama i omogućava konstrukciju SA u linearnom vremenu. U algoritmu SACA-K koristimo inducirano sortiranje za LMS-podnizove, kao i za određivanje SA(S) iz SA(S1).

Općenit algoritam za sortiranje LMS-podnizova je sljedeći:

1. Inicijaliziraj svaki element niza SA[0, n-1] na početnu vrijednost.
2. Prođi kroz S jednom zdesna nalijevo i spremi sve LMS-podnizove od S u pretince koji odgovaraju njihovom početnom znaku, npr. lms(S, i) ide u pretinac bucket(SA, S, i), idući od kraja prema početku svakog pretinca.
3. Prođi kroz SA jednom slijeva nadesno. Za svaki neprazan SA[i], j = SA[i]-1, ako je S[j] L-tip, stavi suf(S, j) na trenutno najljeviju praznu poziciju u pretincu bucket(SA, S, j).
4. Prođi kroz SA jednom zdesna nalijevo. Za svaki neprazan S[i], j = SA[i]-1, ako je S[j] S-tip, stavi suf(S, j) na trenutno najdesniju praznu poziciju u pretincu bucket(SA, S, j).

Općenit algoritam za određivanje SA(S) iz SA(S1) razlikuje se u prva dva koraka:

1. Inicijaliziraj svaki element niza SA[n1, n-1] na početnu vrijednost.
2. Prođi kroz SA[0, n1-1] jednom zdesna nalijevo i spremi sve sortirane LMS-podnizove od S u pretince od SA, idući od kraja prema početku svakog pretinca.
3. Vidi gore.
4. Vidi gore.

U oba slučaja, algoritam induciranog sortiranja radi u složenosti O(n). Međutim, polje B koje sadrži pokazivače na pretince dostupno je samo na dubini 0, stoga ostaju pitanja kako odrediti tip znaka S[j] i kako pratiti koja je trenutno najljevija ili najdesnija pozicija svakog pretinca na dubinama većoj od 0.

2.1.1 Inducirano sortiranje na dubini 0

Budući da na dubini 0 imamo dostupno polje B koje sadrži pokazivače na pretince, možemo ga koristiti slično kao u algoritmu SA-IS. Međutim, i dalje nemamo pomoćno polje P1 koje sadrži početne pozicije LMS-podnizova te sve spremamo u isto polje, SA. U nastavku su prikazani koraci induciranog sortiranja sufiksa na dubini 0.

1. Inicijalizirati svaki element SA[n1, n-1] na početnu vrijednost.
2. Izračunati i postaviti krajnju poziciju svakog pretinca iz SA u polje B[0, K-1]. Proći SA[0, n1-1] zdesna nalijevo i staviti sve sortirane LMS-podnizove iz S u pripadajuće pretince u SA, od kraja prema početku svakog pretinca na sljedeći način: za svaki element SA[i], j = SA[i] i c = S[j], postavi SA[i] na početnu vrijednost, SA[B[c]] = j i smanji B[c] za 1.
3. Izračunati i postaviti početnu poziciju svakog pretinca iz SA u polje B[0, K-1]. Proći SA slijeva nadesno i inducirano sortirati sufikse L-tipa iz S u pripadajuće pretince u SA, od početka prema kraju svakog pretinca na sljedeći način: za svaki neprazan element SA[i], j = SA[i] - 1, c = S[j], ako je S[j] L-tipa, onda postavi SA[B[c]] = j i povećaj B[c] za 1.
4. Izračunati i postaviti krajnju poziciju svakog pretinca iz SA u polje B[0, K-1]. Proći SA zdesna nalijevo i inducirano sortirati sufikse S-tipa iz S u pripadajuće pretince u SA, od kraja prema početku svakog pretinca na sljedeći način: za svaki neprazan element SA[i], j = SA[i] - 1, c = S[j], ako je S[j] S-tipa, onda postavi SA[B[c]] = j i smanji B[c] za 1.

U nastavku je prikazana metoda koji izvršava inducirano sortiranje LMS-podnizova ili sufiksa (bilo L-tipa ili S-tipa, objedinjeno je u jednu metodu) na dubini 0.

```
// Performs induced sorting of LMS-substrings or suffixes (depending on the flag) at
level 0.
// author: mfolnovic
void inducedSort0(uchar* T, uint* SA, uint* bkt, uint K, uint n, bool processing_S_type,
bool suffix) {
    // initialize buckets to start/end of each bucket
    initializeBuckets(T, bkt, K, n, /* set_to_end */ processing_S_type);

    if (!processing_S_type) {
        bkt[0] += 1;
    }

    scan {
        // for each scanned non-empty item SA[i]
        if (SA[i] > 0) {
            uint j = SA[i] - 1;
            uchar c, curr, next;
            c = curr = T[j], next = T[j + 1];

            bool is_S_type = curr <= next && bkt[T[j]] < i;
```

```
            bool is_L_type = curr >= next;

            // if T[j] is L-type/S-type
            if ((processing_S_type && is_S_type) || (!processing_S_type && is_L_type)) {
                // set SA[bkt[c]] = j
                SA[bkt[curr]] = j;
                // ... and increase/decrease bkt[c] by 1
                bkt[c] += processing_S_type ? -1 : 1;

                if (!suffix && (processing_S_type || (!processing_S_type && i > 0)))
SA[i] = 0;
            }
        }
    }
}
```

2.1.2 Inducirano sortiranje na dubini većoj od 0

Budući da na dubini većoj od 0 nemamo pristup pomoćnom polju B koje sadrži pokazivače na pretince, potrebno je smisliti novi način praćenja pretinaca. Srećom, postoji zgodno svojstvo znakova L-tipa koje možemo iskoristiti za konstruiranje SA(S) bez korištenja pomoćnog polja B, a to je:

Na dubini > 0, svaki znak L-tipa ili S-tipa u S pokazuje na početak, odnosno kraj, svog pretinca u SA.

Ideja je iskoristiti početak svakog pretinca u SA kao pokazivač na poziciju gdje bi sufiks L-tipa koji sortiramo u taj pretinac trebao biti pohranjen. Ako želimo inducirano sortirati sve sufikse L-tipa, proći ćemo po SA slijeva nadesno i činiti sljedeće: za svaki SA[i] > 0, j = SA[i] - 1, ako je S[j] L-tip znaka (u ovom slučaju, S[j] je L-tip znaka ako je S[j] >= S[j+1]), onda stavljamo suf(S, j) u pripadajući pretinac u SA. Prema prethodno navedenom svojstvu, S[j] pokazuje na početak svog pretinca u SA. Drugim riječima, ako je c = S[j], početak pretinca bucket(SA, S, j) je SA[c]. Kako bismo zapamtili da element iz SA koristimo kao pokazivač za neki pretinac, postavljamo vrijednost tog elementa na neku nenegativnu vrijednost. Dalje se granamo s obzirom na sljedeće slučajeve:

1. Ako je SA[c] prazan, to znači da je suf(S, j) prvi sufiks koji se postavlja u taj pretinac. U tom slučaju, moramo provjeriti je li SA[c+1] prazan ili nije. Ako je, sortiramo suf(S, j) u SA[c+1] postavljajući SA[c+1] = j i počinjemo koristiti SA[c] kao pokazivač postavljajući SA[c] = -1. Inače, SA[c+1] može biti nenegativan za indeks sufiksa ili negativan za pokazivač, a suf(S, j) mora biti jedini element u svom pretincu, tada stavimo suf(S, j) u svoj pretinac postavljajući SA[c] = j.

2. Ako je SA[c] nenegativan, to znači da je SA[c] “posudio” lijevi susjedni pretinac (bucket(SA, S, j)). U tom slučaju, SA[c] sprema najveći element u lijevom susjednom pretincu te moramo pomaknuti jedan korak ulijevo sve elemente iz lijevog susjednog pretinca kako bi došli na svoje točne pozicije u SA. Početni element lijevog susjednog pretinca možemo naći tako da prolazimo niz od SA[c] ulijevo, dok ne naiđemo na element SA[x] koji je negativan, jer to znači da ga koristimo kao pokazivač. Točnije, x je najveći element za koji vrijedi SA[x[< 0, SA[x] ≠ EMPTY i x < c. Kada nađemo SA[x], pomičemo za jedno mjesto ulijevo sve elemente iz SA[x+1, c] na SA[x, c-1] i postavljamo SA[c] kao prazan element. Sada smo u istoj situaciji kao u koraku 1, što znači da ćemo dalje sortirati suf(S, j) u pripadajući pretinac.
3. Ako je SA[c] negativan i neprazan, to znači da SA[c] koristimo kao pokazivač za bucke(SA, S, j). U tom slučaju, d = SA[c] i pos = c - d + 1, tada je SA[pos] pozicija na koju trebamo smjestiti suf(S, j). Međutim, suf(S, j) može biti najveći sufiks u svojem pretincu. Dakle, moramo provjeriti i vrijednost elementa SA[pos]. Ako je SA[pos] prazan, jednostavno stavljamo suf(S, j) u svoj pretinac tako da postavimo SA[pos] = j i povećamo pokazivač tog pretinca za 1, SA[c] = SA[c]-1 (podsjetnik: SA[c] je negativan). Inače, SA[pos] je početak desnog susjednog pretinca, koji u tom trenutku mora biti nenegativan kao indeks sufiksa ili negativan kao pokazivač. Dakle, moramo pomaknuti za jedno mjesto ulijevo elemente iz SA[c+1, pos-1] u SA[c, pos-2] i zatim sortirati suf(S, j) u svoj pripadajući pretinac tako da postavimo SA[pos-1] = j.

Opisan je algoritam induciranog sortiranja LMS-podnizova i sufiksa L-tipa, no pažljivom implementacijom, metoda za sortiranje sufiksa S-tipa može se objediniti u istu metodu. U nastavku je dana metoda koja implementira inducirano sortiranje na dubini većoj od 0.

```
// Performs induced sorting of LMS-substrings or suffixes (depending on the flag) at
level > 0.
// The parameter processing_type is defined as follows: 0 - LMS, 1 - S-type, 2 - L-type.
// author: pgombar
void inducedSort1(int* T, int* SA, uint n, int processing_type, bool suffix) {
    int step = 1;

    bool processing_LMS = processing_type == 0;
    bool processing_S_type = processing_type == 1;
    bool processing_L_type = processing_type == 2;
    int mul = processing_LMS || processing_S_type ? -1 : 1;

    if (processing_LMS) {
        // initialize each item of SA[0, N-1] as empty
        for (uint i = 0; i < n; i++) {
```

```
            SA[i] = EMPTY;
        }
    }

    bool current_s_type = false;
    scan_complex(processing_S_type, processing_LMS, step) {
        step = 1;
        if (!processing_LMS && SA[i] <= 0) continue;

        uint j;
        int curr;
        bool is_S_type = false, is_L_type = false, is_LMS = false;
        if (processing_LMS) {
            j = i;
            curr = T[j]; int prev = T[j - 1];
            bool previous_s_type = scan_rtl_check_S_type(prev, curr);
            is_LMS = !previous_s_type && current_s_type;
            current_s_type = previous_s_type;
        } else {
            j = SA[i] - 1;
            curr = T[j]; int next = T[j + 1];
            is_S_type = curr < next || (curr == next && curr > (int)i);
            is_L_type = curr >= next;
        }

        if ((processing_LMS && !is_LMS) || (processing_S_type && !is_S_type) ||
            (processing_L_type && !is_L_type)) {
            continue;
        }

        int d = SA[curr];
        // if SA[curr] stores suffix index
        if (d >= 0) {
            // ... then SA[curr] is "borrowed" by the right-neighbouring bucket
            // (of bucket(SA, T, j)). In this case, SA[curr] is storing the smallest
            // item in right-neighbouring bucket, and we need to shift-right one step
            // all the items in the right-neighbouring bucket to their correct locations
            // in SA.
            uint h = shiftValue(SA, curr, -mul, /* check_empty */ !processing_LMS);
            if ((processing_S_type && h > i) || (processing_L_type && h < i)) {
                step = 0;
            }
            d = SA[curr] = EMPTY;
        }

        // if SA[curr] stores empty value...
        if ((uint)d == EMPTY) {
            // ... then suf(T, j) is the first suffix being put into its bucket. In this
case
            //, we further check SA[curr +- 1] to see if it is empty or not. If it is...
            if (((processing_LMS || processing_S_type) && (uint)SA[curr - 1] == EMPTY) ||
                 (processing_L_type && (uint)curr < n - 1 && (uint)SA[curr + 1] ==
EMPTY)) {
                // ... we sort suf(T, j) into SA[curr +- 1] by settings SA[curr +- 1] = j
and
                // start to reuse SA[curr] as a counter by setting SA[curr] = -1.
                SA[curr + mul] = j;
                SA[curr] = -1;
            } else {
                // Otherwise, SA[curr - 1] may be non-negative for a suffix index or
negative
```

```
                // for a counter, and suf(T, j) must be the only element of its bucket,
we hence
                // simply put suf(T, j) into its bucket by settings SA[curr] = j
                SA[curr] = j;
            }
        } else { // else SA[curr] stores bucket counter
            // In this case, let d = SA[curr] and pos = c - d + 1 / c + d - 1, then
SA[pos] is
            // the item that suf(T, j) should be stored into. However, suf(T, j) may be
the
            // smallest/largest suffix in its bucket. Therefore, we further check the
value
            // of SA[pos] to proceed as follows.
            uint pos = curr - mul * (d - 1);
            if ((uint)SA[pos] == EMPTY && (!processing_L_type || pos <= n-1)) {
                // if SA[pos] is empty, we simply put suf(T, j) into its bucket by
setting
                // SA[pos] = j, and increase the counter of its bucket by 1, i.e.
                // SA[curr] = SA[curr] - 1 (notice that SA[curr] is negative for a
counter).
                SA[curr] -= 1;
                SA[pos] = j;
            } else {
                // Otherwise, it indicates that SA[pos] is the start item of the
                // right-neighbouring bucket, which must be currently non-negative for a
                // suffix index or negative for a counter. Hence, we need to
shift-left/right
                // one step the items in SA[pos + 1, curr - 1]/SA[curr + 1, pos - 1] to
                // SA[pos + 2, curr]/SA[curr, pos - 2], then sort suf(T, j) into
                // its bucket by setting SA[pos -+ 1] = j
                shiftCount(SA, curr, -d, mul);
                SA[pos - mul] = j;
                if ((processing_S_type && T[j] > (int)i) || (processing_L_type &&
(uint)curr < i)) {
                    step = 0;
                }
            }
        }

        bool is_L_type1 = (j+1 < n-1) && (T[j+1]>T[j+2] || (T[j+1] == T[j+2] && T[j+1] <
i));
        if ((processing_L_type && (!suffix || !is_L_type1) && i > 0) ||
            (processing_S_type && !suffix)) {
            SA[step == 0 ? i - mul : i] = EMPTY;
        }
    }

    if (processing_LMS || processing_L_type || !suffix) {
        // scan to shift-right the items in each bucket
        // with its head being reused as a counter
        processing_S_type |= processing_LMS;
        scan {
            int j = SA[i];
            if (j < 0 && (uint)j != EMPTY) { // SA[i] stores bucket counter
                //printf("%d %d %d\n", i, -j, mul, i + -j * mul);
                shiftCount(SA, i, -j, mul);
                SA[i + -j * mul] = EMPTY;
            }
        }
    }
```

```
    if (processing_LMS) {
        SA[0] = n - 1;
    }
}
```

2.3 Imenovanje LMS-podnizova

Novost u odnosu na postojeće SACA algoritme koju uvodi Nong u svom radu (Nong et. al, 2011) je upravo drukčiji način imenovanja LMS-podnizova, koji omogućava konstrukciju SA bez upotrebljavanja pomoćnog polja t, koje pamti informacije o tipu znaka, S-tip ili L-tip. Imenovanje LMS-podnizova slijedi nakon sortiranja istih, s ciljem produciranja skraćenog niza S1. Taj skraćeni niz S1 je upravo ulaz rekurzivnog poziva metode SACA-K. U nastavku je opisan način imenovanja LMS-podnizova.

Definirajmo prvo s-rank i se-rank člana niza S. S-rank elementa S[i] je broj elemenata u S manjih od S[i], a se-rank od S[i] je broj elemenata u S manjih ili jednakih S[i] (bez samog elementa S[i]). Pretpostavimo da su svi LMS-podnizovi od S sortirani u SA1, iskoristit ćemo nov način imenovanja LMS-podnizova, kako bismo dobili niz S1 u složenosti O(n). Koraci imenovanja opisani su u nastavku.

1. Proći SA slijeva nadesno i proglasiti da početna pozicija pretinca podniza u SA1 predstavlja ime svakog LMS-podniza u S, rezultirajući privremenim skraćenim nizom Z1, u kojem svaki element pokazuje na početak svog pripadajućeg pretinca u SA1.
2. Proći Z1 zdesna nalijevo i zamijeniti svaki element S-tipa u Z1 do krajnje pozicije pripadajućeg pretinca u SA1, rezultirajući novim nizom S1. Pogodnost u ovom slučaju je ta da odmah možemo saznati kojeg je tipa koji element u Z, jer prolazimo nizom zdesna nalijevo.

Za razliku od imenovanja u algoritmu SA-IS, gdje ime LMS-podniza predstavlja indeks pripadajućeg pretinca u SA1, ovim načinom smo postigli da je u novom nizu S1 svaki znak L-tipa ili S-tipa također s-rank, odnosno se-rank, u S1. Ovime smo postigli da u skraćenom nizu S1 svaki znak L-tipa ili S-tipa pokazuje na početak, odnosno kraj, pripadajućeg pretinca u SA1. U nastavku je prikazana metoda koja obavlja imenovanje LMS-podnizova.

```
// Produces names for LMS-substrings of T to get reduced string T1.
// author: mfolnovic
uint computeLexicographicNames(uchar* T, uint* T1, uint* SA, uint n, uint m, uint n1, int
```

```
level) {
    // init
    for (uint i = n1; i < n; i++) {
        SA[i] = EMPTY;
    }

    // scan SA_1 once from left to right to name each LMS-substring of T by the
    // start position of the substring's bucket in SA_1, resulting in an interim
    // reduced string denoted by Z_1 (where each character points to the start
    // of its bucket in SA_1)
    int previous_lms_length = 0, previous_x = 0;
    int current_name = 0, n_names = 0;

    for (uint j = 0; j < n1; j++) {
        int x = SA[j];
        int lms_length = 1;

        // traverse the LMS-substring from its first character T[x] until we
        // see a character T[x + i] less than its preceding T[x + i - 1]. Now,
        // T[x + i - 1] must be L-type.
        uint i;
        for (i = 1; x + i < n && convert(x + i) >= convert(x + i - 1); i++);

        // Continue to traverse the remaining characters of the LMS-substring
        // and terminate when we see a character T[x + i] greater than its
        // preceding T[x + i - 1] or T[x + i] is the sentinel. At this point,
        // we know that the start of the succeeding LMS-substring has been
        // traversed and its position was previously recorded when we saw
        // T[x + i] < T[x + i - 1] the last time.
        for (; x + i <= n && convert(x + i) <= convert(x + i - 1); i++) {
            if (x + i == n - 1 || convert(x + i) < convert(x + i - 1)) {
                lms_length = i + 1;
            }
        }

        // now determine if current LMS-substring is different than the last one
        bool is_different = false;
        if (lms_length != previous_lms_length) is_different = true;
        else {
            for (int offset = 0; offset < lms_length && !is_different; offset++) {
                uint current_pos = x + offset;
                uint previous_pos = previous_x + offset;
                is_different = current_pos == n - 1 || previous_pos == n - 1 ||
                    convert(current_pos) != convert(previous_pos);
            }
        }
        if (is_different) {
            // it's different so create new name...
            current_name = j;
            n_names += 1;
            SA[current_name] = 1;

            // it's different so we'll compare next LMS-substring with this one
            previous_x = x;
            previous_lms_length = lms_length;
        } else {
            // it's same, so reuse the name
            SA[current_name] += 1;
        }

        SA[n1 + x/2] = current_name;
```

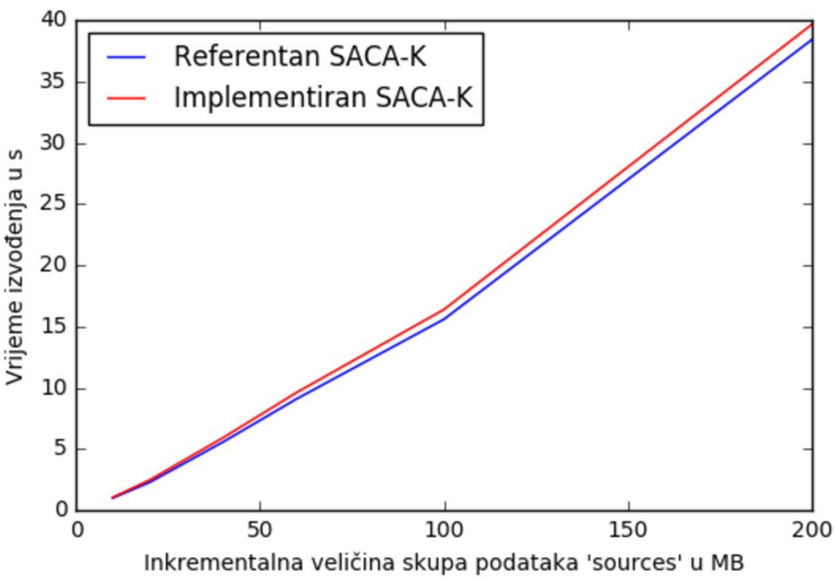
```
    }

    // compact...
    for (uint i = n - 1, j = m - 1; i >= n1; i--)
        if (SA[i] != EMPTY) SA[j--] = SA[i];

    // Scan Z_1 once from right to left to replace each S-type character in Z1
    // by the end position of its bucket in SA_1, resulting in the new string T1.
    bool current_s_type = false;
    for (int i = n1-1; i > 0; i--) {
        bool previous_s_type = scan_rtl_check_S_type(T1[i - 1], T1[i]);

        // if S-type
        if (previous_s_type) {
            T1[i - 1] += SA[T1[i - 1]] - 1;
        }

        current_s_type = previous_s_type;
    }

    // return number of unique LMS substrings (or number of names...)
    return n_names;
}
```

3. Usporedba performansi

Testiranje vremenskih i memorijskih performanski provedeno je nad nekoliko skupova podataka. Sva su mjerenja izvršena na prijenosnom računalu s 16 GB RAM-a i procesorom Intel(R) Core(TM) i7-4700MQ CPU @ 2.40GHz. Detalji o skupovima podataka su prikazani u sljedećoj tablici.

Skup podataka	**Veličina skupa podataka**	**Opis**
sources	210.9 MB	Svi izvorni kodovi C/Java programa iz distribucija linux-2.6.11.6 i gcc-4.0.0.
kernel	258 MB	Skup verzija 1.0.x i 1.1.x Linux kernela.
fib41	267.9 MB	Fibonaccijev niz.
Salmonella Enterica	4.9 MB	Genom Salmonelle Enterice.
Escherichia Coli	5.4 MB	Genom Escherichie Coli.
synthetic.1e5	100 kB	Sintetički generirani podaci.
synthetic.1e6	1 MB	Sintetički generirani podaci.

U nastavku je prikazano vrijeme izvođenja referentne implementacije SACA-K algoritma te implementiranog SACA-K algoritma nad skupom podataka `sources`.



Također, dostupna su vremena izvođenja referentne implementacije te naše implementacije na svim ostalim skupovima podataka, kao i veličina korištenog radnog prostora.

Skup podataka	**Referentno vrijeme izvođenja**	**Implementirano vrijeme izvođenja**	**Veličina korištenog radnog prostora**	**Veličina skupa podataka**
sources	**33.131 s**	35.835 s	1007 MB	210.9 MB
kernel	**44.579 s**	46.007 s	1232 MB	258 MB
fib41	48.814 s	**47.984 s**	1279 MB	267.9 MB
Salmonella Enterica	**0.438 s**	0.460 s	25 MB	4.9 MB
Escherichia Coli	**0.499 s**	0.518 s	27 MB	5.4 MB
synthetic.1e5	0.010 s	**0.007 s**	2 MB	100 kB
synthetic.1e6	0.090 s	**0.081 s**	6 MB	1 MB

4. Zaključak

Sufiksno polje korisna je struktura podataka koja omogućuje indeksiranje i pretraživanje nizova. Polazeći od algoritma za izgradnju sufiksnog polja SA-IS, koji pruža linearnu složenost izgradnje, memorijski učinkovitu izvedbu i jednostavnu implementaciju, algoritam SACA-K uvodi poboljšanja te se pokazao 33% bržim i koristi manji radni prostor, ovisan samo o veličini ulazne abecede.

Pokazali smo da se implementirana verzija SACA-K algoritma može mjeriti s referentnom implementacijom po pitanju vremenskog izvođenja i memorijske potrošnje. Na skupu podataka `sources`, naša implementacija je tek neznatno sporija od referentne implementacije, dok obje implementacije koriste jednaku veličinu radnog prostora nad svim skupovima podataka.

5. Literatura

1. Gusfield, Dan. Algorithms on strings, trees and sequences: computer science and computational biology. Cambridge university press, 1997.
2. Ukkonen, Esko. "On-line construction of suffix trees." Algorithmica 14.3 (1995): 249-260.
3. Wu, Sun, et al. "An O (NP) sequence comparison algorithm." Information Processing Letters 35.6 (1990): 317-323.
4. Abouelhoda, Mohamed Ibrahim, Stefan Kurtz, and Enno Ohlebusch. "Replacing suffix trees with enhanced suffix arrays." Journal of Discrete Algorithms 2.1 (2004): 53-86.
5. Nong, Ge, Sen Zhang, and Wai Hong Chan. "Linear suffix array construction by almost pure induced-sorting." 2009 Data Compression Conference. IEEE, 2009.
6. Nong, Ge, Sen Zhang, and Wai Hong Chan. "Two efficient algorithms for linear time suffix array construction." IEEE Transactions on Computers 60.10 (2011): 1471-1484.
7. Nong, Ge. "Practical linear-time O (1)-workspace suffix sorting for constant alphabets." ACM Transactions on Information Systems (TOIS) 31.3 (2013): 15.
8. Hadviger, Antea. Poboljšano sufiksno polje / završni rad - preddiplomski studij. Zagreb, Fakultet elektrotehnike i računarstva, 3.6. 2015, 41 str. Voditelj: Šikić, Mile
9. Šikić, Mile, Domazet-Lošo, Mirjana. Bioinformatika / skripta. Zagreb, Fakultet elektrotehnike i računarstva, prosinac 2013.