Izvedba algoritama sufiksnog stabla u funkcijskome programskom jeziku Haskell

Jan Šnajder
Zavod za elektroniku, mikroelektroniku, računalne i inteligentne sustave
Fakultet elektrotehnike i računarstva Sveučilišta u Zagrebu
Unska 3, 10000 Zagreb, Hrvatska
jan.snajder@fer.hr

25. studenog 2008.

Sažetak

Podatkovna struktura sufiksnog stabla omogućava pretraživanje nad nizom znakova u vremenu proporcionalnom duljini traženog uzorka. U radu su opisane četiri programske izvedbe algoritma izgradnje sufiksnog stabla u čistom funkcijskom jeziku Haskell. Dvije su izvedbe konceptualno funkcijske, dok se dvije oslanjaju na monadičko programiranje i konceptualno su imperativne. Pored analize asimptotske vremenske složenosti ovih algoritama, dana je i usporedba njihovih empirijskih složenosti.

1 Uvod

Sufiksno stablo je podatkovna struktura koja na kompaktan način pohranjuje sve sufikse nekog znakovnog niza. Sufiksna se stabla mogu koristiti za pronalaženje uzorka teksta u vremenu proporcionalnom duljini traženog uzorka, neovisno o duljini samog teksta. Zbog toga su sufiksna stabla posebno prikladna za učestala pretraživanja kraćih uzoraka u duljim tekstovima, kao na primjer pri analizi genomskih sekvenci u bioinformatici. Algoritmi za izgradnju i pretraživanje sufiksnih stabala detaljno su opisani [2].

U ovome radu opisana je izvedba algoritama izgradnje sufiksnog stabla u funkcijskom programskom jeziku Haskell [4].[1] Funkcijski jezici temeljeni su na formalizmu lambda računa i deklarativnom pristupu programiranju. To je osobito slučaj kod *čisto funkcijskih* (engl. purely functional) jezika kao što Haskell. Kod takvih jezika izvođenje programa, odnosno izračunavanje programskih izraza, ne smije izazvati nikakve popratne učinke jer bi oni mogli narušiti tzv. referencijalnu transparentnost tih izraza. Izravna prednost ovakvog pristupa jest mogućnost (formalnog) rasuđivanja o svojstvima programa. Osim što je čisto funkcijski, Haskell je i *lijeni* funkcijski jezik, što znači da izračunavanje izraza odgađa sve do trenutka dok vrijednost tog izraza nije potrebna.

[1] Izvorni kôd dostupan je na adresi: `http://www.zemris.fer.hr/~jan/haskell/SuffixTree.hs`

Lijena evaluacija u načelu omogućava učinkovitije izvođenje programa, a omogućava i ostvarivanje beskonačnih podatkovnih struktura. Zainteresirani čitatelj upućuje se na [3], gdje je dan izvrstan pregled koncepata funkcijskih jezika.

Rad je u nastavku organiziran na sljedeći način. U dijelu 2 opisana su dva funkcijska algoritma izgradnje sufiksnog stabla. U dijelu 3 opisana su dva algoritma koji su izvedbeno funkcijski, ali konceptualno imperativni. Rezultati mjerenja vremena izvođenja dani su u dijelu 4.

2 Funkcijski algoritmi

2.1 Predstavljanje sufiksnog stabla i teksta

Tekst u Haskellu najjednostavnije bismo mogli prikazati nizom znakova (stringom), odnosno općenito listom vrijednosti. Ako bismo na isti način prikazali i oznake na bridovima, tada bi prostorna složenost stabla bila neprihvatljivih $O(n^n)$, gdje je n duljina teksta. Linearnu prostornu složenost možemo vrlo jednostavno ostvariti tako da umjesto nizova znakova za oznake bridova koristimo indekse podnizova u tekstu. U tom slučaju potrebno je također osigurati da se pojedinim znakovima teksta može pristupiti u konstantnom vremenu. Zbog toga ćemo za prikaz teksta umjesto listi koristiti polje:

```
type Text a = Array Int a
```

Zbog općenitosti, tip `Text` ovdje je definiran kao polimorfan tip parametriziran tipskom varijablom `a`; tip `a` jest tip vrijednosti koje sačinjavaju niz (za tekstovne nizova to će biti tip `Char`). Tip `Array` je Haskellov ugrađeni tip za *nepromjenjiva* (engl. immutable) polja, koji ne podržava promjenu jednom postavljenih vrijednosti pojedinih elemenata, što međutim ovdje niti nije potrebno. Pretvorbu slijeda predstavljenog listom `[a]` u vrijednost tipa `Text a` provodi sljedeća funkcija:

```
text :: [a] -> Text a
text t = listArray (1,length t) t
```

Na primjer:

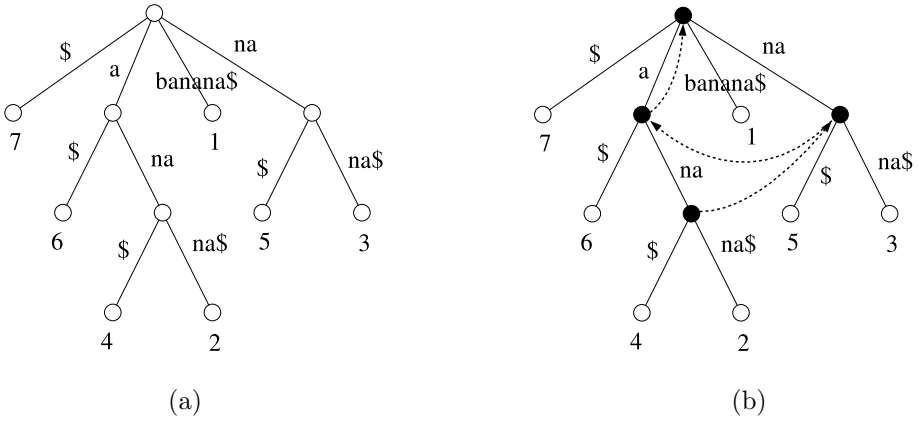
```
> let t = text "banana$"
array (1,7)[(1,'b'),(2,'a'),(3,'n'),(4,'a'),(5,'n'),(6,'a'),(7,'$')]
```

Uz ovako prikazani tekst, sufiks možemo jednostavno predstaviti dvojkom (p, q), gdje je p pozicija početnog, a q pozicija završnog znaka sufiksa:

```
type Suffix = (Int,Int)
```

Sufikse teksta duljine n generira sljedeće funkcija:

```
suffixes :: Text a -> [Suffix]
suffixes t = [(i,n) | i <- [1..n]]
  where (_,n) = bounds t
```



Slika 1: Sufiksno stablo za niz "banana$" (a) i isto stablo sa sufiksnim vezama (b).

Primjerice:

```
> suffixes t
[(1,7),(2,7),(3,7),(4,7),(5,7),(6,7),(7,7)]
```

Struktura stabla, a i mnoge druge strukture, kod funkcijskih se jezika, odnosno općenito kod deklarativnih jezika, uobičajeno predstavljaju rekurzivnom podatkovnom strukturom. Za strukturu sufiksnog stabla definirat ćemo u Haskellu poseban (rekurzivan) tip podataka:

```
data STree = Leaf | Branch [(Label, STree)] deriving (Eq, Show)
type Label =  Suffix
```

Definicija kazuje da je `STree` ili završni čvor (tipski konstruktor `Leaf`) ili unutarnji čvor (tipski konstruktor `Branch`). Kod unutarnjeg čvora pohranjujemo listu dvojki `(Label,STree)`, gdje je `Label` oznaka brida (dvojka početne i završne pozicije podniza u slijedu) dok je `STree` podstablo do kojega brid vodi. Primjerice, sufiksnom stablu za niz "banana$", prikazanom slikom 1 (a), odgovara sljedeća rekurzivna struktura:

```
Branch [
  ((7,7),Leaf),
  ((2,2),Branch [
    ((7,7),Leaf),
    ((3,4),Branch [
      ((7,7),Leaf),((5,7),Leaf)])]),
  ((1,7),Leaf),
  ((3,4),Branch [
    ((7,7),Leaf),
    ((5,7),Leaf)])]
```

2.2 Lijena izgradnja sufiksnog stabla

Lijenim postupkom izgradnje sufiksnog stabla smatramo onaj potupak koji stablo gradi inkrementalno od korijena prema završnim čvorovima [1]. Prednost takva pristupa jest ta što se izgradnja stabla provodi usporedo s njegovim obilaskom, pa će pojedini putevi u stablu biti konstruirani tek kada za to postoji potreba. U lijenim funkcijskim jezicima kao što je Haskell takva se vrsta inkrementalnosti ostvaruje automatski.

Na slici 2 prikazan je algoritam lijene izgradnje sufiksnog stabla temeljen na algoritmu predloženom u [1]. Počevši od korijena, algoritam najprije grupira sufikse prema početnome slovu te za svaku takvu grupu sufikasa kreira odgovarajući brid i odgovarajuće podstablo. Postupak se zatim rekurzivno ponavlja sve dok ne preostane samo jedan sufiks, kada se kreira odgovarajući završni čvor. Za odabir sufiksa prema zadanom početnom slovu koristi se funkcija `select` služi za odabir sufikasa prema zadanom početnom slovu. Na primjer:

```
> select t 'n' $ suffixes t
[(3,7),(5,7)]
```

Funkcija `edge` za danu grupu sufikasa kreira oznaku brida kao njihov najdulji zajednički prefiks te listu ostataka sufikasa, tj. sufikasa koji se dobivaju uklanjanjem spomenutog prefiksa. Na primjer:

```
> edge t [(3,7),(5,7)]
((3,4),[(5,7),(7,7)])
```

Glavna funkcija `lazyTree` pri izgradnji unutarnjeg čvora iterira kroz sva slova abecede `alph` te za svako slovo za koje funkcija `select` vrati nepraznu listu kreira odgovarajući brid i odgovarajuće podstablo. (Ovdje je spretno iskorištena tzv. *list comprehension* notacija kako bi se izbjeglo korištenje lokalnih `let` definicija, a isto je tako vješto iskorišten mehanizam podudaranja uzorka kako bi se implicitno provjerilo vraća li funkcija `select` nepraznu listu.) Izgradnju sufiksnog stabla za tekst "banana$" demonstrira sljedeći poziv funkcije:

```
> let alph = '$':['a'..'z'] in lazyTree alph t
Branch [((7,7),Leaf),((2,2),Branch [((7,7),Leaf),((3,4),Branch [((7,7),Leaf),
((5,7),Leaf)])]),((1,7),Leaf),((3,4),Branch [((7,7),Leaf),((5,7),Leaf)])]
```

Za određivanje asimptotske vremenske složenosti funkcije `lazyTree`, bitna je činjenica da će u konačnici funkcija `edge` morati učitati svaki pojedini znak svakog sufiksa. To za tekst duljine n daje $\frac{n}{2}(n+1)$ operacija čitanja znaka. Pored toga, funkcija `lazyTree` u najgorem slučaju svaki učitani znak mora usporediti s k znakova abecede. Složenost funkcije `lazyTree` stoga je u najgorem slučaju $O(kn^2)$.

2.3 Izgradnja stabla umetanjem

Algoritam umetanja sufikasa, koji je u osnovi pojednostavljena varijanta McCreightovog algoritma [6], jednostavan je i intuitivan način izgradnje sufiksnog stabla. Počevši od

```
lazyTree :: (Eq a) => [a] -> Text a -> STree
lazyTree alph t =
  node $ suffixes t
  where node [] = Leaf
        node ss = Branch [((p,q), node ssr) | a <- alph,
                                              ass@(_:_) <- [select t a ss],
                                              ((p,q),ssr) <- [edge t ass]]

select :: (Eq a) => Text a -> a -> [Suffix] -> [Suffix]
select t a ss = [ (p,q) | (p,q) <- ss, t!p == a]

edge :: (Eq a) => Text a -> [Suffix] -> (Label, [Suffix])
edge _ [s] = (s,[])
edge t ass@((p,q):ss)
  | all sameFirst ss = ((p,q2),rss)
  | otherwise        = ((0,p-1),ass)
  where ((_,q2),rss) = edge t [(p+1,q) | (p,q) <- ass, p < q]
        sameFirst (r,_) = t!p==t!r
```

Slika 2: Algoritam lijene izgradnje sufiksnog stabla.

najduljeg sufiksa, sufiksi se jedan po jedan umeću u sve veće stablo. Svaki se sufiks umeće krenuvši od korijenskog čvora, i to tako da se prefiks sufiksa uspoređuje s oznakama na bridovima sve dok postoji podudaranje. Pri prvom nepodudaranju dolazi do grananja u stablu. Nepodudaranje može nastupiti na već postojećem unutarnjem čvoru, ali i na bilo kojem mjestu unutar brida, kada je brid potrebno razlomiti uvođenjem novog unutarnjeg čvora.

Izvedba algoritma u Haskellu, preuzeta uz manje preinake iz [1], dana je na slici 3. Funkcija `cplen` koristi se za izračun duljine podudarnog prefiksa dvaju sufikasa. Na primjer:

```
> cplen t (2,7) (4,7)
3
```

Funkcija `naiveInsert` temelji se na iterativnoj primjeni funkcije `insertSuffix` koja služi umetanju jednog sufiksa. Iteriranje je ostvareno funkcijskim obrascom "lijevog presavijanja" `foldl`, odnosno njegovom striktnom i time prostorno učinkovitijom varijantom `foldl'`. Funkcija `insertSuffix`, sastavljena je od pomoćne funkcije `ins`, obilazak stabla čini usporedo s njegovom modifikacijom.

Početno umetanje započinje praznim stablom `Branch []` i ono jednostavno stvara brid koji završava završnim čvorom. Kod svih daljnih umetanja moguća su četiri slučaja. Ako ne postoji brid koji se u prvome znaku podudara s novim sufiksom, onda je čvoru potrebno dodati novi brid i to, ovisno o leksikografskom poretku, ili s lijeva ili s desna.

```
naiveInsert :: (Ord a) => Text a -> STree
naiveInsert t = foldl' (insertSuffix t) (Branch []) (suffixes t)

insertSuffix :: (Ord a) => (Text a) -> STree -> Suffix -> STree
insertSuffix t (Branch es) s@(p1,q1) =
  Branch (ins es)
  where ins [] = [(s,Leaf)]
        ins bs@(b@(e@(p2,q2),node):es)
          | t!p1 > t!p2    = b:ins es
          | t!p1 < t!p2    = (s,Leaf):bs
          | y > q2         = (cp,insertSuffix t node ex):es
          | cpl == p1-q1+1 = b:es
          | t!x < t!y      = (cp,Branch [nx,ny]):es
          | otherwise      = (cp,Branch [ny,nx]):es
          where cpl = cplen t s e
                cp = (p1,p1+cpl-1)
                (x,y) = (p1+cpl,p2+cpl)
                (ex,ey) = ((x,q1),(y,q2))
                nx = (ex,Leaf)
                ny = (ey,node)

cplen :: (Eq a) => Text a -> Suffix -> Suffix -> Int
cplen t (p1,q1) (p2,q2) =
  until (\i-> i>=to || t!(p1+i)/=t!(p2+i)) (+1) 1
  where (l1,l2) = (q1-p1+1,q2-p2+1)
        to = min l2 l1
```

Slika 3: Izgradnja sufiksnog stabla umetanjem sufikasa.

Ako, međutim, takav brid već postoji, moguća su tri slučaja. Prvo, brid je posve podudaran s prefiksom novog sufiksa, i u tom slučaju spuštamo se niz brid i funkcija `insertSuffix` rekurzivno se poziva za ostatak sufiksa. Drugo, ako je duljina podudarnog prefiksa manja od duljine brida, onda novi sufiks implicitno već postoji u sufiksnom stablu i ne treba činiti ništa. Treće, novi sufiks i brid ne podudaraju se u cijelosti, i u tom je slučaju brid potrebno razlomiti novim unutarnjim čvorom. Taj se čvor s lijeva grana na novi završni čvor, a s desna na postojeće podstablo, odnosno obrnuto, ovisno o leksikografskom poretku. Izgradnju sufiksnog stabla za slijed "banana$" demonstrira sljedeći poziv:

```
> naiveInsert $ text
Branch [((7,7),Leaf),((6,6),Branch [((7,7),Leaf),((5,6),Branch [((7,7),Leaf),
((5,7),Leaf)])]),((1,7),Leaf),((5,6),Branch [((7,7),Leaf),((5,7),Leaf)])]
```

Valja ukazati na to da je dobiveno stablo izomorfno s onim dobivenim primjenom funkcije `lazyTree`, ali da postoje razlike u oznakama na bridovima.

Analiza asimptotske složenosti algoritma `naiveInsert` je sljedeća. Za slijed duljine n potrebno je umetnuti ukupno n sufikasa, a svako umetanje zahtijeva spuštanje od korijenskog čvora niz put čija je duljina proporcionalna s n. Pored toga, odabir podudarnog brida u svakom unutranjem čvoru uvodi faktor k, gdje je k broj znakova abecede. Asimptotska složenost algoritma `naiveInsert` stoga je u najgorem slučaju $O(kn^2)$.

2.4 Proširenje oznaka na bridovima

Numeričke oznake na bridovima omogućuju značajne memorijske uštede, međutim one stablo čine teže čitljivim. Pored toga, budući da se jedno te isti podniz u slijedu može pojavljivati više puta, numeričke oznake nisu jedinstvene, što onemogućava provjeru ekvivalentnosti sufiksnih stabala izgrađenih različitim algoritmima. Zbog toga ćemo definirati funkciju za proširenje oznaka na bridovima, odnosno za zamjenu numeričkih oznaka odgovarajućim tekstnim nizovima. Sufiksno stablo s proširenim oznakama bridova definiramo na sljedeći način:

```
data STreeE a = LeafE | BranchE [([a], STreeE a)] deriving (Eq,Show)
```

Tip `STreeE` istovjetan je tipu `STree`, s tom razlikom da je za oznake bridova korištena polimorfna lista `[a]`. Pretvorbu sufiksnog stabla tipa `STree` u sufiksno stablo tipa `STreeE` obavlja funkcija `expandEdges` sa slike 4. Na primjer, proširenje oznaka sufiksnog stabla za slijed "banana$" rezultira sljedećim:

```
> expandEdges t $ naiveInsert t
BranchE [("$",LeafE),("a",BranchE [("$",LeafE),("na",BranchE [("$",LeafE),
("na$",LeafE)])]),("banana$",LeafE),("na",BranchE [("$",LeafE),("na$",LeafE)])]
```

Također, uporabom funkcije `expandEdges` možemo provjeriti ekvivalentnost dvaju sufiksnih stabala:

```
> let alph = '$':['a'..'z'] in
  expandEdges t (naiveInsert t) == expandEdges t (lazyTree alph t)
True
```

```
expandEdges :: (Eq a) => Text a -> STree -> STreeE a
expandEdges t (Branch es) =
  BranchE [([t!i | i <- [p..q]],expandEdges t node) | ((p,q),node) <- es]
expandEdges _ Leaf = LeafE
```

Slika 4: Proširenje oznaka na bridovima.

```
addPositions :: (Eq a) => Text a -> STree -> STreeP
addPositions t node = add node 0
  where add Leaf d = LeafP (n-d+1)
        add (Branch es) d =
          BranchP [((p,q),add node (d+q-p+1)) | ((p,q),node) <- es]
        (_,n) = bounds t
```

Slika 5: Dodavanje vrijednosti početnih pozicija u završne čvorove.

2.5 Pozicijsko sufiksno stablo

Sufiksno stablo definirano podatkovnim tipom `STree` ne sadržava informaciju o početnim pozicijama pojedinih sufikasa u tekstu. Budući da postoji jedan-na-jedan korespondencija između sufikasa i završnih čvora sufiksnog stabla, takvu je informaciju moguće pohraniti u završne čvorove. Definirajmo najprije odgovarajuću podatkovnu strukturu:

```
data STreeP = LeafP Int | BranchP [(Label, STreeP)] deriving (Eq,Show)
```

Ova je struktura istovjetna strukturi `STree`, s tom razlikom da u završnim čvorovima pohranjuje vrijednost početne pozicije sufiksa, koja je tipa `Int`. Dodavanje te informacije u završne čvorove, odnosno pretvorbu strukture tipa `STree` u strukturu tipa `STreeP` obavlja funkcija `addPositions` sa slike 5. Funkcija je definirana rekurzivno nad strukturom tipa `STree`, a pozicije u završnim čvorovima izračunava tako da dubinu svakog završnog čvora (izraženu brojem znakova na bridovima počevši od korijena) oduzme od ukupne duljine teksta n. Na primjer, dodavanje početnih pozicija sufikasa u sufiksno stablo za slijed "banana$" rezultira sljedećim pozicijskim stablom:

```
> addPositions t $ naiveInsert t
BranchP [((7,7),LeafP 7),((6,6),BranchP [((7,7),LeafP 6),((5,6),BranchP [((7,7),
LeafP 4),((5,7),LeafP 2)])]),((1,7),LeafP 1),((5,6),BranchP [((7,7),LeafP 5),
((5,7),LeafP 3)])]
```

```
search :: (Eq a) => Text a -> STreeP -> [a] -> [Int]
search _ (LeafP i) [] = [i]
search _ (LeafP _)  _ = []
search t (BranchP es) [] = concat [search t node [] | (_,node) <- es]
search t (BranchP es) s@(a:_) =
  case find (\((p,_),_) -> t!p == a) es of
    Just ((p,q),node) -> if (s1 == e) then search t node s2 else []
                         where l = min (length s) (q-p+1)
                               (s1,s2) = splitAt l s
                               e = [t!i | i <- [p..p+l-1]]
    _                 -> []
```

Slika 6: Pretraživanje sufiksnog stabla.

2.6 Pronalaženje podniza

Osnovna primjena sufiksnog stabla jest pronalaženje podnizova (uzoraka) u tekstu. Jednom kada je sufiksno stablo izgrađeno, pretraživanje uzorka provodi se na sljedeći način. Krenuvši od korijena, znakovi uzorka uspoređuju se s bridovima sufiksnog stabla sve dok ne dođe do nepodudaranja ili dok se ne potroše svi znakovi uzorka. U prvome slučaju, uzorak se u tekstu ne pojavljuje. U drugome slučaju, uzorak se pojavljuje u tekstu, a završni čvorovi ispod mjesta posljednjeg podudaranja odgovaraju svim njegovim početnim pozicijama u tekstu.

Opisanu funkcionalnost implementira funkcija `search` sa slike 6. Funkcija kao ulaz uzima tekst, pozicijsko sufiksno stablo te traženi uzorak, a vraća listu početnih pozicija uzorka u tekstu, odnosno praznu listu ako uzorak nije pronađen. U sufiksnom se stablu najprije pronalazi brid koji se s traženim uzorkom podudara u početnome znaku, a zatim uspoređuje traženi uzorak i oznaku na tom bridu. Ako takav brid ne postoji, ili ako se uzorak i oznake na bridu ne podudaraju, funkcija vraća praznu listu. U suprotnom, uzorak se skraćuje i postupak se rekurzivno nastavlja sve dok se uzorak ne potroši. Ako se uzorak potroši na mjestu unutarnjeg čvora, stablo se rekurzivno obilazi u širinu (engl. breadth first). Kada se dosegne završni čvor, u njemu pohranjena vrijednost početne pozicije sufiksa prenosi se u listu rezultata. Ako se završni čvor dosegne prije nego što se uzorak potroši, to znači da je uzorak prefiks nekog sufiksa te da u tekstu nije sadržan, pa funkcija tada također vraća praznu listu. Na primjer, pretraživanje teksta “banana$” za pojavljivanjem podniza “an” rezultira sljedećim:

```
> let st = addPositions t $ naiveInsert t in search t st "na"
[5,3]
```

međutim

```
> let st = addPositions t $ naiveInsert t in search t st "nab"
[]
```

Treba naglasiti da se funkciju `search` može napraviti i tako da ona izravno pretražuje sufiksno stablo tipa `STree`, tj. stablo koje u završnim čvorovima nema pohranjene vrijednosti početnih pozicija sufikasa, budući da se te vrijednosti ionako mogu izračunati prilikom obilaska stabla. Međutim, u kontekstu Haskellove lijene evaluacije, svejedno je jesu li funkcije `addPositions` i `search` združene u jednu, ili se primijenjuju u kompoziciji – zbog lijene evaluacije one će se ionako izvoditi isprepleteno. U tom smislu, razdvajanje funkcionalnosti u dvije zasebne funkcije i uvođenje dodatnog tipa podataka `STreeP` čini se konceptualno boljim rješenjem.

2.7 Dodavanje sufiksnih veza

Funkcije `lazyTree` i `naiveInsert` sufiksno stablo izgrađuju ne koristeći tzv. *sufiksne veze*. Sufiksna veza povezuje unutarnji čvor koji odgovara sufiksu $x\alpha$ s unutarnjim čvorom koji odgovara sufiksu α, gdje je x neki znak, a α (moguće prazan) slijed znakova. Može se dokazati (v. npr. [6]) da će svaki unutarnji čvor sufiksnog stabla (osim korijena) biti izvorištem sufiksne veze. Dodavanjem sufiksnih veza sufiksno se stablo zapravo pretvara u ciklički digraf; sufiksno stablo teksta "banana$" sa sufiksnim vezama prikazano je slikom 1 (b). Kako je istaknuto u [1], sufiksne su veze potrebne za neke primjene i zato je bitno demonstrirati kako se sufiksnim vezama može proširiti struktura tipa `STree`, unatoč tome što je ta struktura bitno drugačija od one imperativne koja je tipično sagrađena od međusobno povezanih čvorova. Premda kod rekurzivno definirane strukture `STree` sufiksne veze nije moguće ostvariti uporabom pokazivača, moguće je rekurzivno povezati dva podstabla koja bi inače bila povezana sufiksnim vezama. Definirajmo najprije novu rekurzivnu strukturu `STreeL`:

```
data STreeL = LeafL | BranchL [(Label, STreeL)] STreeL | Null
  deriving (Eq, Show)
```

Sufiksna veza jest vrijednost tipa `STreeL` pohranjena u unutarnjem čvoru. Ako sufiksna veza nije definirana, koristi se vrijednost `Null`. Bitno je naglasiti da, budući da je sufiksno stablo sa sufiksnim vezama ciklički digraf, to je struktura `STreeL` beskonačna. To ujedno znači da ovaj pristup nužno iziskuje lijeni funkcijski jezik poput Haskella; kod striktne evaluacije pokušaj izgradnje beskonačne strukture ne bi se nikada zaustavio.

Pretvorbu sufiksnog stabla tipa `STree` u sufiksno stablo sa sufiksnim vezama `STreeL` ostvaruje funkcija `addLinks` sa slike 7, preuzeta, uz manje izmjene, iz [1]. Funkcija je definirana rekurzivno nad strukturom tipa `STree`. Funkcija `down` vraća podstablo krenuvši od zadanog čvora i spuštajući se niz zadanu stazu do unutarnjeg čvora. U funkciji `addLinks` rekurzivno se izgrađuje sufiksno stablo sa sufiksnim vezama. Pomoćna funkcija `link` rekurzivno postavlja sufiksne veze svim unutarnjim čvorovima osim korijenskom. Pritom se odgovarajuće podstablo (ono na koje upućuje sufiksna veza) pronalazi funkcijom `down`. Kada se postavljaju sufiksne veze djeci korjenskoga čvora, funkcija `down` podstablo pronalazi krenuvši od sufiksa s odbačenim prvim znakom, dok se kod svih idućih razina kreće od podstabla `l` na koje upućuje sufiksna veza. Budući da funkcija `addLinks` sufiksne veze postavlja jednim obilaskom stabla, asimpotska složenost funkcije

```
addLinks :: (Eq a) => Text a -> STree -> STreeL
addLinks t (Branch es) = rootL
  where rootL = BranchL [((p,q),link (down t rootL (p+1,q)) node) |
                           ((p,q),node) <- es] Null
        link _ Leaf = LeafL
        link l (Branch es) =
          BranchL [(cp,link (down t l cp) node) | (cp,node) <- es] l

down :: (Eq a) => Text a -> STreeL -> Suffix-> STreeL
down t node@(BranchL es _) (p1,q1)
  | p1 <= q1  = down t node (p1+cpl,q1)
  where (cpl,node) = head [(q2-p2+1,node) | ((p2,q2),node) <- es, t!p1==t!p2]
down _ node _ = node
```

Slika 7: Dodavanje sufiksnih veza.

`addLinks` je, u najgorem slučaju, $O(n)$, gdje je n duljina teksta (funkcija `down` ne ovisi o n).

Za ilustraciju, dodajmo veze sufiksnom stablu za tekst "banana$". Budući da je rezultirajuća struktura beskonačna, nije ju moguće cijelu niti izračunati niti ispisati. Možemo, međutim, ispis ograničiti na prvih, recimo, 400 znakova:

```
> take 400 $ show $ addLinks t $ naiveInsert t
"BranchL [((7,7),LeafL),((6,6),BranchL [((7,7),LeafL),((5,6),BranchL [((7,7),
LeafL),((5,7),LeafL)] (BranchL [((7,7),LeafL),((5,7),LeafL)] (BranchL [((7,7),
LeafL),((5,6),BranchL [((7,7),LeafL),((5,7),LeafL)] (BranchL [((7,7),LeafL),
((5,7),LeafL)] (BranchL [((7,7),LeafL),((5,6),BranchL [((7,7),LeafL),((5,7),
LeafL)] (BranchL [((7,7),LeafL),((5,7),LeafL)] (BranchL [((7,7),LeafL),((5,6),
BranchL [((7,7),"
```

3 Imperativni algoritmi

Uz iznimku gore opisanih algoritama preuzetih iz [1], u literaturi se razmatraju isključivo imperativni algortmi izgradnje sufiksnog stabla. To uključuje i poznati Ukkonenov inkrementalni algoritam s linearnom vremenskom složenošću [8]. Imperativni algoritmi oslanjaju se na prikaz sufiksnog stabla kao skupa međusobno povezanih čvorova u pohranjenih dinamičkoj memoriji, te na mogućnost (destruktivne) izmjene bilo kojeg dijela te strukture. Postojanje strukture u dinamičkoj memoriji koju algoritmi mogu izravno mijenjati implicira izračunavanje sa stanjem (engl. statefull computation), odnosno postojanje popratnih učinaka (engl. side-effects). Budući da (čisto) funkcijski jezici ne dopuštaju popratne učinke, takvi se jezici nerijetko smatraju manje učinkovitim od imperativnih [7]. Zbog toga je interesantno razmotriti kako se konceptualno imperativni

algoritmi poput Ukkonenovog mogu ostvariti u čisto funkcijskom jeziku kao što je Haskell. To zatim omogućava da se učinkovitost funkcijskih i imperativnih algoritama izravno usporedi.

3.1 Monadičko programiranje

Haskell je čisto funkcijski jezik i kao takav ne dopušta baš nikakve popratne učinke kojima bi se eventualno narušila referencijalna transparentnost. Kako je, međutim, stvaranje popratnih učinaka suština nekih vrlo temeljnih operacija, poput ulazno-izlaznih operacija ili operacije generiranja slučajnih brojeva, jezik mora pružati odgovarajuću alternativu koja će ga učiniti praktički upotrebljivim. Haskell takvu alternativu pruža u obliku tzv. *monada*. Najjednostavnije rečeno, monade omogućavaju da se izračunavanje sa stanjem provodi bez ikakvih popratnih učinaka. Monadičko programiranje, dakle, omogućava da program mijenjanja svoje implicitno stanje bez narušavanja referencijalne transparentnosti. Čitatelj zainteresiran za monadičko programiranje upućuje se na neki od udžbenika o Haskellu, npr. [4].

3.2 Dinamička memorija ostvarena u monadi IO

Imperativni algoritama izgradnje sufiksnog stabla oslanjaju se mogućnost stvaranja čvorova u dinamičkoj memoriji. Sličnu funkcionalnost možemo ostvariti u Haskellu uporabom monadičkog programiranja. Pritom ćemo se ograničiti na monadu `IO` kojom se u Haskellu ostvaruju sve ulazno-izlazne operacije, te na vrlo pojednostalvjenu, ali za naše potrebe dostatnu izvedbu dinamičke memorije.

Dinamičku memoriju definirat ćemo kao dvojku sastavljenu od polimorfnog polja koje pohranjuje vrijednosti tipa `a` te cjelobrojnog indeksa koji odgovara prvom nepopunjenom zapisu tog polja:

```
type Heap a = (IOArray Int (Maybe a),IORef Int)
```

Oba elementa ove dvojke pripadaju monadi `IO`. Tip `IOArray` je polje kojemu je moguće pristupati i koje je moguće mijenjati u konstantnom vremenu. Elementi polja vrijednosti su tipa `Maybe`, što nam omogućava da još nepopunjeni zapis predstavimo vrijednošću `Nothing`. Tip `IORef Int` je referenca na najmanji indeks nepopunjenog zapisa; koristimo referencu kako bismo tu vrijednost mogli mijenjati. Pokazivač na vrijednost pohranjenu u memoriji definiramo kao

```
type Pointer a = Int
nil = 0 :: Pointer a
```

i ona odgovara indeksu odgovarajućeg zapisa u polju. Vrijednost `nil` koristimo kao indikaciju da pokazivač nije definiran.

Funkcije za manipulaciju ovako definiranom dinamičkom memorijom dane su na slici 8. Izvedba bi bila nešto složenija ako bismo željeli omogućiti oslobađanje zapisa dinamičke memorije, međutim to nam ovdje neće biti potrebno. Funkciju `getHeap`,

```
createHeap :: Int -> IO (Heap a)
createHeap n =
  do arr <- newArray (1,n) Nothing
     iRef <- newIORef 1
     return (arr,iRef)

new :: Heap a -> a -> IO (Pointer a)
new (arr,iRef) v =
  do i <- readIORef iRef
     writeArray arr i $ Just v
     modifyIORef iRef (+1)
     return i

get :: Heap a -> Pointer a -> IO a
get (arr,_) p = do Just v <- readArray arr p
                   return v

set :: Heap a -> Pointer a -> a -> IO ()
set (arr,_) p v = writeArray arr p $ Just v

getHeap :: Heap a -> IO [Maybe a]
getHeap (arr,_) = getElems arr
```

Slika 8: Manipulacija dinamičkom memorijom u monadi IO.

```
intoSTree :: Heap Node -> (Pointer Node) -> IO STree
intoSTree heap p = get heap p >>= \node -> recurse heap node
  where recurse _ (LeafH i) = return $ Leaf i
        recurse heap (BranchH es _) =
          do es' <- forM es (\(el,p) -> intoSTree heap p >>=
                               \node -> return $ (el,node))
             return $ Branch es'
```

Slika 9: Pretvorba imperativnog sufiksnog stabla u rekurzivnu strukturu.

koja sadržaj dinamičke memorije pretvara u listu, koristit ćemo za provjeru sadržaja memorije. Konačno, čvor sufiksnog stabla pohranjen u dinamičkoj memoriji definiramo na sljedeći način:

```
data Node = LeafH Int | BranchH [(Label, Pointer Node)] (Pointer Node)
  deriving (Show, Eq)
```

Za razliku od ranije definiranih tipova sufiksnog stabla, ovaj tip nije rekurzivan. Kod unutarnjih čvorova (tipski konstruktor `BranchH`) definiramo i pokazivač sufiksne veze. Korisnim će se pokazati pretvorba ovako definiranog sufiksnog stabla u rekurzivnu strukturu stabla tipa `STree`. Tu pretvorbu obavlja funkcija `intoSTree` dana slikom 9.

3.3 Imperativni algoritam izgradnje stabla umetanjem

Imperativna varijanta algoritma umetanja sufikasa prikazana je slikom 10. Algoritam je izravna reimplementacija funkcijske varijante sa slike 3. Funkcije `changeEdge` i `insertEdge` koriste se za promjenu brida unutarnjeg čvora (u slučaju kada je brid potrebno razlomiti), odnosno za dodavanje novog brida unutarnjem čvoru. Funkcija `impNaiveInsert` kao ulaz uzima dinamičku memoriju tipa `Heap Node` i tekst, a vraća pokazivač na korijenski čvor. Sufiksne veze u ovom algoritmu se ne koriste. Sufiksno stablo za tekst "banana$" izgradit ćemo na sljedeći način:

```
> do h <- createHeap 10; impNaiveInsert h t
1
```

Uvid u sadržaj dinamičke memorije dobit ćemo na sljedeći način:

```
> do h <- createHeap 15; impNaiveInsert h t; getHeap h
[Just (BranchH [((7,7),11),((2,2),10),((1,7),2),((3,4),8)] 0),Just LeafH,
Just LeafH,Just LeafH,Just LeafH,Just (BranchH [((7,7),5),((5,7),3)] 0),
Just LeafH,Just (BranchH [((7,7),7),((5,7),4)] 0),Just LeafH,Just (BranchH
[((7,7),9),((3,4),6)] 0),Just LeafH,Nothing,Nothing,Nothing,Nothing]
```

dok pretvorbu u rekurzivnu strukturu tipa `STree` ovako:

```
impNaiveInsert :: (Ord a) => Heap Node -> Text a -> IO (Pointer Node)
impNaiveInsert h t =
  do root <- new h (BranchH [] nil)
     forM_ (suffixes t) $ impInsertSuffix h t root
     return root

impInsertSuffix :: (Ord a) => Heap Node -> Text a -> Pointer Node -> Suffix
                              -> IO ()
impInsertSuffix h t pNode s@(p1,q1) =
  do (BranchH es l) <- get h pNode
     case find (\((p2,_),_)-> t!p1==t!p2) es of
       Just e@((p2,q2),pChild) ->
         do let (l1,l2) = (q1-p1+1,q2-p2+1)
                cpl = cplen t (p1,q1) (p2,q2)
            child <- get h pChild
            if l1 <= cpl then
              return ()
              else if cpl == l2 then
                impInsertSuffix h t pChild (p1+cpl,q1)
                else
                do pNewLeaf <- new h LeafH
                   let (e1,e2) = (((p1+cpl,q1),pNewLeaf),((p2+cpl,q2),pChild))
                       es' = if t!(p1+cpl) < t!(p2+cpl) then [e1,e2] else [e2,e1]
                   pNewNode <- new h (BranchH es' nil)
                   changeEdge h t pNode e ((p2,p2+cpl-1),pNewNode)
       Nothing -> do pNewLeaf <- new h LeafH
                     insertEdge h t pNode ((p1,q1),pNewLeaf)

changeEdge h t pNode e1 e2 =
  do (BranchH es l) <- get h pNode
     let (es1,(_:es2)) = break (==e1) es
     set h pNode $ BranchH (es1 ++ (e2:es2)) l

insertEdge h t pNode e@((p,_),_) =
  do (BranchH es l) <- get h pNode
     let  (es1,es2) = break (\((r,_),_) -> t!r > t!p) es
     set h pNode $ BranchH (es1 ++ (e:es2)) l
```

Slika 10: Imperativna verzija algoritma umetanja sufikasa.

```
> do h <- createHeap 15; r <- impNaiveInsert h t; intoSTree h r
Branch [((7,7),Leaf),((2,2),Branch [((7,7),Leaf),((3,4),Branch [((7,7),Leaf),
((5,7),Leaf)])]),((1,7),Leaf),((3,4),Branch [((7,7),Leaf),((5,7),Leaf)])]
```

Također se možemo uvjeriti da je funkcijski i imperativni algoritam daju identičan rezultat:

```
> do h <- createHeap 15; r <- impNaiveInsert h t; st1 <- intoSTree h r;
    let st2 = naiveInsert t in return $ expandEdges t st1 == expandEdges t st2
True
```

3.4 Algoritam inkrementalne izgradnje sufiksnog stabla

Sva tri do sada opisana algoritma izgradnje sufiksnog stabla kvadratične su vremenske složenosti s obzirom na duljinu teksta. Razvijeni su, međutim, i algoritmi s linearnom složenoću, od kojih se konceptualno najjednostavnijim smatra Ukkonenov algoritam [8]. Dodatna prednost Ukkonenova algoritma jest njegova inkrementalnost, u smislu da algoritam sufiksno stablo gradi postepeno, čitajući znakove teksta s lijeva na desno.

Ukkonenov algoritam stablo izgrađuje kroz niz *faza*, svaka od kojih se sastoji od određenog broja *ekstenzija*. Za tekst $x_1, \ldots, x_n$, algoritam će sufiksno stablo izgraditi kroz n faza. U i-toj fazi, $1 \leq i \leq n$, izgrađeno je sufiksno stablo za text $x_1, \ldots, x_i$. U $i+1$ fazi, sufiksno se stablo proširuje tako da ono sadržava sve sufikse podniza $x_1, \ldots, x_{i+1}$, kojih je ukupno $i+1$. Dodavanje tih sufikasa odvija se kroz $i+1$ ekstenzija: j-ta ekstenzija faze $i+1$ u sufiksno stablo dodaje podniz $x_j, \ldots, x_{i+1}$, gdje $1 \leq j \leq i+1$. Dodavanje novog sufiksa svodi se na jednu od dvije akcije: produljivanje završnog čvora ili dodavanje novog brida (bilo u točci unutarnjeg čvora ili lomljenjem postojećeg brida). Ako sufiks implicitno već postoji u stablu, ne čini se ništa.

Vremenska složenost opisanog postupaka, kada bi se taj implementirao jednostavnim umetanjem sufikasa, bila bi $O(n^3)$. Ključ za ubrzanje algoritma jest u tome da se u fazi $i+1$ omogući brzo pronalaženje kraja svakog od ukupno $i+1$ sufikasa podniza $x_1, \ldots, x_{i+1}$. Ukkonenov algoritam u tu svrhu koristi sufiksne veze. Sufiksne veze, uz tri dodatna jednostavna trika, Ukkonenovom algoritmu omogućavaju izgradnju sufiksnog stabla s asimpotskom vremenskom složenošću $O(n)$. Čitatelj zainteresiran za detaljniji opis Ukkonenovog algoritma upućuje se na [2].

Izvedba Ukkonenovog algoritma iz [8] u Haskellu dana je na slici 11. Funkcije `canonize` provodi kanonizaciju tzv. referentnog para dok funkcija `testAndSet` za zadani referentni par i ulazni znak ispituje je li sufiksno stablo potrebno proširivati, odnosno može li se faza raznije prekinuti, te eventualno stvara novi unutarnji čvor. Funkcija `update` implementira jednu fazu, odnosno iterira kroz ekstenzije. Funkcija `onlineTree` najprije kreira korijenski čvor i pomoćni čvor (pomoćni čvor koristi se u [8] radi pojednostavljenja implementacije) te zatim iterira kroz faze učitavajući znak za znakom ulaznog teksta. Pomoćna funkcija `setLink` sa slike 12 služi postavljanju sufiksne veze prethodno stvorenom čvoru.

Na primjer, Ukkonenov algoritam za tekst "banana$" izgradit će u dinamičkoj memoriji sljedeće sufiksno stablo:

```
onlineTree :: (Ord a) => Heap Node -> Text a -> IO (Pointer Node)
onlineTree h t =
  do root <- new h $ BranchH [] 2
     aux <- new h $ BranchH [((0,0),root)] nil
     foldM_ (\(pNode,p1) i -> do (pNode2,p2) <- update h t (pNode,(p1,i))
                                 canonize h t (pNode2,(p2,i))) (root,1) [1..n]
     return root
  where (_,n) = bounds t

update h t rp@(pNode,(p1,q1)) =
  do let pLastNode = 1
     tas <- testAndSplit h t (pNode,(p1,q1-1)) q1
     (endPoint@(pNode,_),pLastNode) <- update' rp tas pLastNode
     when (pLastNode /= 1) $ setLink h pLastNode pNode
     return endPoint
  where update' rp@(pNode,(p1,q1)) (isEndPoint,pNewNode) pLastNode
          | isEndPoint = return ((pNode,p1),pLastNode)
          | otherwise  = do pNewLeaf <- new h LeafH
                            insertEdge h t pNewNode ((q1,n),pNewLeaf)
                            when (pLastNode /= 1) $ setLink h pLastNode pNewNode
                            (BranchH _ link) <- get h pNode
                            (pNode2,p2) <- canonize h t (link,(p1,q1-1))
                            tas <- testAndSplit h t (pNode2,(p2,q1-1)) q1
                            update' (pNode2,(p2,q1)) tas pNewNode
        (_,n) = bounds t

testAndSplit h t rp@(pNode,(p1,q1)) i
  | p1 <= q1 =
    do (BranchH es l) <- get h pNode
       let Just e@((p2,q2),pChild) = find (\((p2,_),_)-> p2==0 || t!p1==t!p2) es
       if t!(p2+q1-p1+1)==t!i then return (True,pNode)
         else do pNewNode <- new h $ BranchH [((p2+q1-p1+1,q2),pChild)] nil
                 changeEdge h t pNode e ((p2,p2+q1-p1),pNewNode)
                 return (False, pNewNode)
  | otherwise =
    do (BranchH es l) <- get h pNode
       return (isJust $ find (\((p2,_),_)-> p2==0 || t!i == t!p2) es, pNode)

canonize h t (pNode,(p1,q1))
  | p1 > q1   = return (pNode,p1)
  | otherwise =
    do (BranchH es l) <- get h pNode
       let Just ((p2,q2),pChild) = find (\((p2,_),_)-> p2==0 || t!p1==t!p2) es
           (l1,l2) = (q1-p1+1,q2-p2+1)
       if l1 >= l2 then canonize h t (pChild,(p1+l2,q1)) else return (pNode,p1)
```

Slika 11: Ukkonenov inkrementalni algoritam izgradnje sufiksnog stabla.

```
setLink :: Heap Node -> Pointer Node -> Pointer Node -> IO ()
setLink h pNode1 pNode2 =
  do (BranchH es l) <- get h pNode1
     set h pNode1 (BranchH es pNode2)
```

Slika 12: Postavljanje sufiksne veze.

```
> let t = text "banana$" in do h <- createHeap 15; onlineTree h t >> getHeap h
[Just (BranchH [((7,7),12),((2,2),10),((1,7),3),((3,4),8)] 2),
Just (BranchH [((0,0),1)] 0),Just LeafH,Just LeafH,Just LeafH,
Just (BranchH [((7,7),7),((5,7),4)] 8),Just LeafH,Just (BranchH [((7,7),9),
((5,7),5)] 10),Just LeafH,Just (BranchH [((7,7),11),((3,4),6)] 1),
Just LeafH,Just LeafH,Nothing,Nothing,Nothing]
```

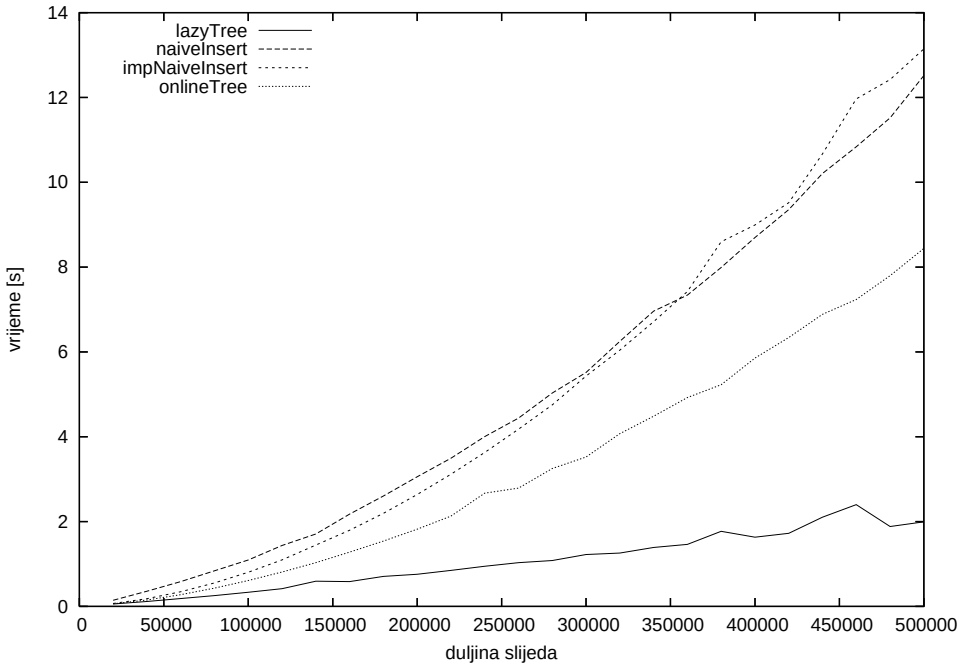
Dobiveno stablo odgovara onome sa slike 1 (b) (čvor s indeksom 2, na koji upućuje sufiksna veza iz korijenskoga čvora s indeksom 1, je pomoćni i može se zanemariti).

4 Eksperimentalni rezultati

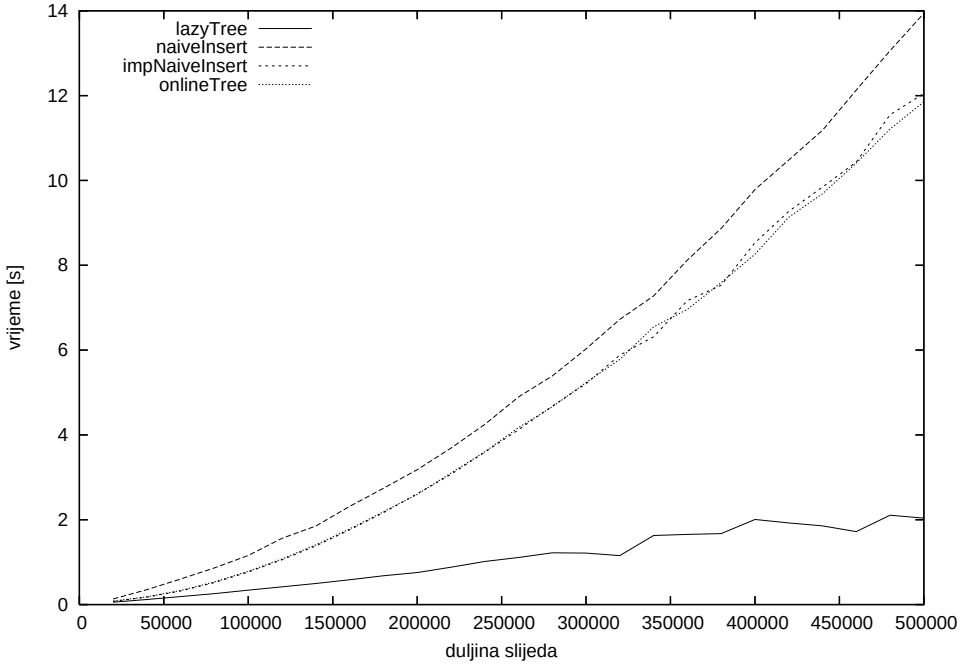
S ciljem utvrđivanja empirijske vremenske složenosti opisanih algoritama, provedeno je mjerenje stvarnog vremena potrebnog za izgradnju sufiksnog stabla. Mjerenje je provedeno nad slučajno generiranim nizom čija je duljna n varirana od od 2000 do $500,000$ znakova u koracima od 2000. Nad svakim su nizom ispitana sva četiri algoritma, te je za svaki algoritam uzeta vrijednost medijana od pet mjerenja. Rezultati mjerenja za veličinu abecede k od 4, 20 i 50 znakova prikazani su slikama 13, 14, odnosno 15. Izvršni kôd programa dobiven je prevodiocem GHC,[2] a mjerenje je provedeno na računalu s Intelovim procesorom takta 2 GHz i 2 GB radne memorije.

Rezultati pokazuju da se za duljinu teksta do $n = 500,000$ sva četiri algoritma ponašaju linearno, neovisno o njihovoj asimptotskoj vremenskoj složenosti. Algoritam lijene izgradnje sufiksnog stabla `lazyTree` pokazao se najučinkovitijim, što je možda iznenađujuće, ali je u skladu sa sličnim ispitivanjem provedenim u [1]. Najlošijim se pokazao funkcijski algoritam umetanja sufikasa `naiveInsert`. Za abecede veličine $k = 20$ i $k = 50$ učinkovitost Ukkonenovog algoritma `onlineTree` ne razlikuje se značajno od učinkovitosti imperativnog algoritma umetanja sufikasa `impNaiveInsert`.

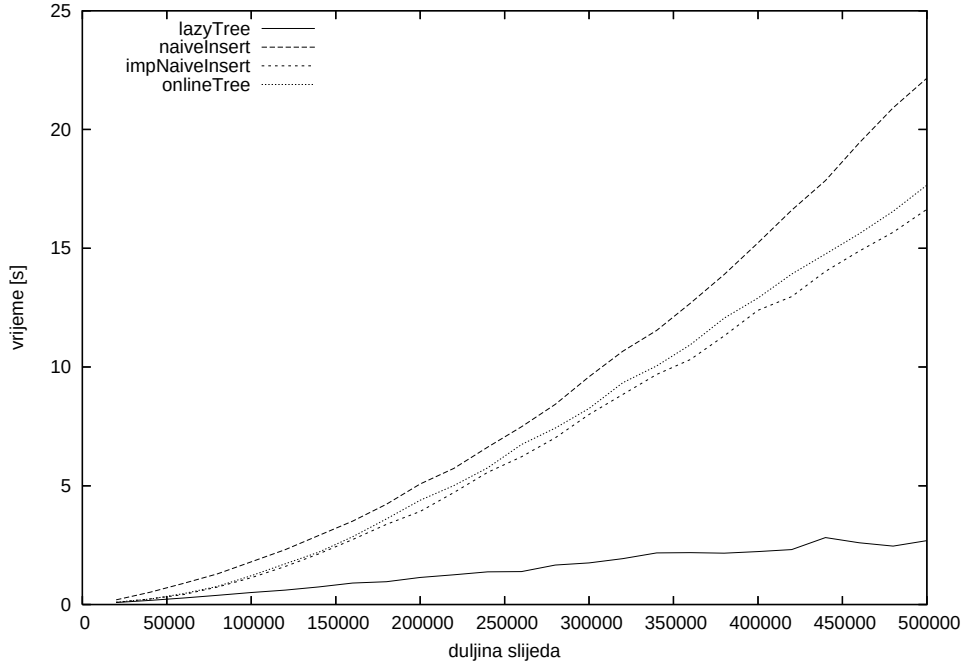
[2] Glasgow Haskell Compiler, verzija 6.8.2; `http://www.haskell.org/ghc/`



Slika 13: Vremena izvođenja za $k = 4$.



Slika 14: Vremena izvođenja za $k = 20$.



Slika 15: Vremena izvođenja za $k = 50$.

Literatura

[1] R. Giegerich, S. Kurtz. A Comparison of Imperative and Purely Functional Suffix Tree Constructions. Science of Computer Programming, 25(2–3):187–218, 1995.

[2] D. Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, 1997.

[3] P. Hudak. Conception, Evolution, and Application of Functional Programming Languages. *ACM Computing Surveys*, 21(3):359–411, 1989.

[4] P. Hudak. *The Haskell School of Expression.* Cambridge University Press, 2000.

[5] S. P. Jones. *Haskell 98 language and libraries: The revised report.* `http://www.haskell.org/definition`.

[6] E. M. McCreight. A Space-Economical Suffix Tree Construction Algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[7] C. G. Ponder, P. C. McGeer, A. P-C. Ng. Are Applicative Languages Inefficient? *SIGPLAN Notices*, 6:135–139, 1988.

[8] E. Ukkonen. On-line Construction of Suffix-Trees. On-line Construction of Suffix Trees. *Algorithmica*, 14(3):249–260.