Yuliya Duniak, Kompiuterinis modeliavimas, 1 kursas

Sufiksų medis yra labai svarbi ir naudinga daugelio programų duomenų struktūra. Tradiciniai tokio medžio konstravimo algoritmai labai priklauso nuo to, ar visos priesagos yra įterptos, kad būtų gautos efektyvios laiko ribos. Todėl straipsnio tikslas buvo sukurti tokią duomenų struktūrą, kuri leidžia atlikti paieškos operacijas su viena prieiga prie antrinės saugyklos, naudojant tik O (m) pirminio saugojimo langelius, nepriklausomai nuo paieškos eilutės ilgio.

Autoriai straipsnyje nagrinėjo tris algoritmus, dviejų iš jų pagrindas buvo algoritmas Ukkonen‘o, o pirmas pristatytas algoritmas net ne toks efektyvus, kaip Ukkonen‘o. Tam, kad suprasti, kaip gi vis dėl to veikia sufiksų medis ir kodėl autoriai ieškojo efektyvesnio būdo sudaryti medį, buvo nuspręsta realizuoti optimizuotą Ukkonen algoritmą bei įvertinti jo laiko ir vietos sudėtingumus.

Taigi, tam, kad įgyvendinti algoritmą buvo panaudoti tokie pagalbiniai kintamieji kaip:

1. *activeEdgeIndex* - parodantis, iš kur turime pradėti dėti naują priesagą.
2. *offset* - rodo priesagų, kurias dar turime praeiti, skaičių.

Ir tokios funkcijos kaip:

1. *getChildEdgeIndex()* – vaiko indekso gavimui;
2. *comeDown()* – leistis per medį;
3. *putEdge()* – paskutinės įterptos briaunos indekso grąžinimui;
4. *putChar()* – pagrindinei algoritmo logikai aprašyti;
5. *edge()* – briaunos gavimui;
6. *print()* – medžio spausdinimui;
7. *printTree()* – gauto medžio formavimui ir paruošimui GraphvizOnline įrankiui atvaizduoti.

Taigi pradžiai įterpiame simbolį į masyvą ir padidiname skaitiklį. Sukuriame vietą sufiksų nuorodoms laikyti, tam kad žymėti panašias briaunas vieno simbolio ribose. Didiname anksčiau minėtą skaitiklį, kadangi atsirado dar vienas simbolis eilutėje. Tuomet, tol, kol nepraeiti visi eilutės simboliai, atliekame veiksmus, laikantys šių taisyklių:

1. Jei po įterpimo iš aktyvaus mazgo = šaknies aktyvusis ilgis yra didesnis nei 0, tada aktyvus mazgas nekeičiamas, aktyvus ilgis sumažėja, aktyvus kraštas paslinktas į dešinę (iki kitos priesagos pirmojo ženklo, kurį turime įterpti)
2. Jei mes sukursime naują mazgą arba iš mazgo padarysime intarpą, ir tai nėra pirmas toks mazgas dabartiniame žingsnyje, tada ankstesnį tokį mazgą susiejame su šiuo per priesagos nuorodą.
3. Po įterpimo iš aktyvaus mazgo, kuris nėra šakninis mazgas, turime sekti priesagos nuorodą ir nustatyti aktyvųjį mazgą prie jo nurodyto mazgo. Jei nėra priesagos nuorodos, nustatome aktyvųjį mazgą į šaknies mazgą. Bet kuriuo atveju aktyvus kraštas ir aktyvus ilgis nesikeičia.

Žemiau esantis pseudokodas aiškiau atvaizduoja pačio algoritmo veikimo principą:

Algorithm 3: The algorithm by Ukkonen.

input : string X of length N
output: suffix tree ST of X

```
1  initialize ST with the empty root node;
2  initialize activePoint=root;
3  for i = 1, ··· , N do
4  |  ST=updateTree(ST with activePoint, prefix P_i);
5  end
6  return ST;
```

Function updateTree(ST, P_i)

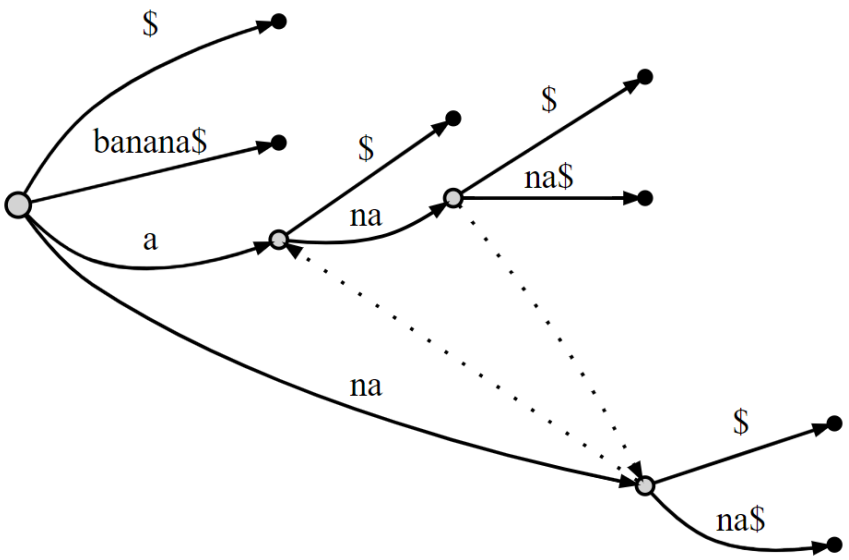
input : tree ST for prefix P_{i-1} with $activePoint$, current prefix P_i
output: tree ST for prefix P_i

```
1  currentSufEnd = activePoint;
2  currentChar = X[i];
3  done=false;
4  while not done do
5  |  if currentSufEnd is at explicit node v then
6  |  |  if v has no child starting with currentChar then
7  |  |  |  create new leaf;
8  |  |  end
9  |  |  else
10 |  |  |  advance activePoint down the corresponding edge;
11 |  |  |  done = true;
12 |  |  end
13 |  end
14 |  else
15 |  |  if the implicit node's next char is not equal currentChar then
16 |  |  |  create new explicit node;
17 |  |  |  create new leaf;
18 |  |  end
19 |  |  else
20 |  |  |  advance activePoint down the corresponding edge;
21 |  |  |  done = true;
22 |  |  end
23 |  end
24 |  if currentSufEnd is at root then
25 |  |  done = true;
26 |  end
27 |  else
28 |  |  proceed to the next smaller suffix following a suffix link ;
29 |  |  currentSufEnd = NextSmallerSuffix ;
30 |  end
31 end
32 activePoint = currentSufEnd;
33 return ST;
```

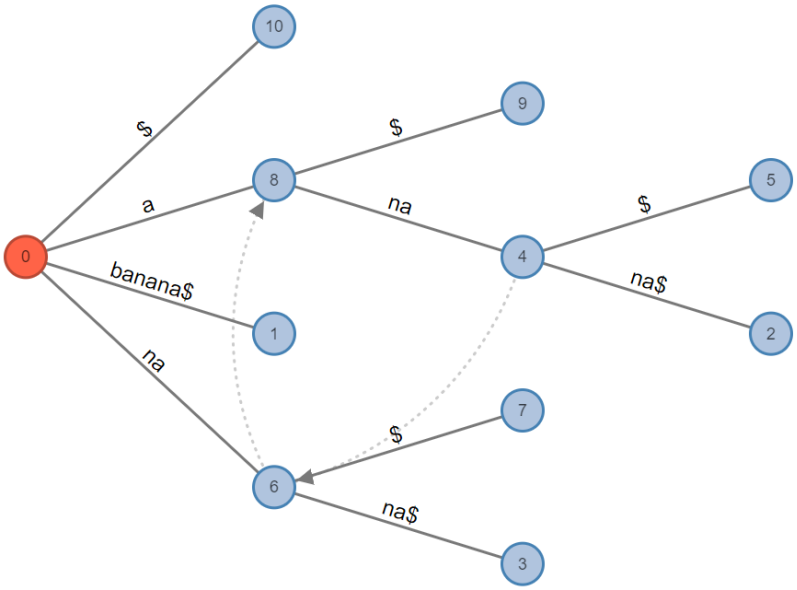
Visų žingsnių pabaigoje *printTree()* ir *print()* funkcijų dėka, įvedę eilutę „banana“ turėtumėme gauti tokį rezultatą:

```
digraph {
rankdir = LR;
edge [arrowsize=0.4,fontsize=10]
node0
[label="",style=filled,fillcolor=lightgrey,shape=circle,width=.1,height=.1]
;
node10[shape=point]
node9[shape=point]
node5[shape=point]
node2[shape=point]
node1[shape=point]
node7[shape=point]
node3[shape=point]
node8[label="",style=filled,fillcolor=lightgrey,shape=circle,width=.07,heig
ht=.07]
node4[label="",style=filled,fillcolor=lightgrey,shape=circle,width=.07,heig
ht=.07]
node6[label="",style=filled,fillcolor=lightgrey,shape=circle,width=.07,heig
ht=.07]
node0 -> node10[label="$",weight=3]
node0 -> node8[label="a",weight=3]
node8 -> node9[label="$",weight=3]
node8 -> node4[label="na",weight=3]
node4 -> node5[label="$",weight=3]
node4 -> node2[label="na$",weight=3]
node0 -> node1[label="banana$",weight=3]
node0 -> node6[label="na",weight=3]
node6 -> node7[label="$",weight=3]
node6 -> node3[label="na$",weight=3]
node4 -> node6 [weight=1,style=dotted]
node6 -> node8 [weight=1,style=dotted]
}
```

Šią dalį įstačius į GraphvizOnline įrankį gauname štai tokį sufiksų medį:



Tam, kad patikrinti, ar gautas rezultatas yra tinkamas, savo medį lyginau su kito įrankio sugeneruotu medžiu (čia).



Taigi šio darbo rezultatas - realizuotas Ukkonen algoritmas, kuriuo kiekvieno ciklo „prasukimo“ metu atliekamas darbas yra O(1), nes visos briaunos atnaujinamos automatiškai, didinant skaičių. Taip pat ir vieno papildomo simbolio įterpimas vyksta tik per O (1) laiką. Kadangi, mūsų ciklą praeina kiekvienas eilutės simbolis, algoritmas veikia tiesiniu laiku ir, galima manyti, kad n sudėtingumo medžiui sukurti reikia tik O (n) laiko ir O (m) vietos.