Семинарски рад:

Суфиксна стабла

Марко Протулипац

Садржај:

Увод..........................................................................................................3

Шта је суфиксно стабло.........................................................................4

Реализација..............................................................................................4

Pravila za produženje sufiksa ...................................................................................6

Убрзавање алгоритма..............................................................................................7

-Суфиксни линкови..........................................................................................................7

-Прескочи трик..................................................................................................................8

- Алгоритам за појединачно продужење.......................................................................9

-Сузбијање........................................................................................................................10

-Убрзање 3. правилом.....................................................................................................10

-Једном лист увек лист....................................................................................................11

Kod………………………………………………………………………………..11

Увод:

Један од најобимнијих пројеката у биологији представља проучавање генома живих врста. Биолози су почели да добијају запаженије резултате на овом пољу тек након увођења компјутера у рад приликом њиховог истраживања, а њихови резултати су се побољшавали упоредо са развојем биоинформатике као посебне гране информатике. Проучавање генома је у највећој мери засновано на претраживању стрингова и налажењу одређених узорака у оквиру њих. Самим тим један од основних императива биоинформатике је био развој конкретних алата односно алгоритама за претраживање стрингова.

Временом су се као најефикасније наметнуле две структуре података за ову сврху - суфиксна стабла и суфиксни низови. Неумољива жеђ научника за комплетирањем знања о геномима живих врста, а самим тим и проницање у тајне живота је била главна покретачка сила за развој све ефикаснијих алгоритама за рад са стринговима. Ти алгоритми су даље послужили као основа за даљи развој модерних софтверских апликација који савременим биолозима омогућују неупоредиво брже и лакше истраживање генома.

Тема рада који је пред вама су основни алгоритми за конструкцију суфиксних стабала и суфиксних низова и примене ових структура података. Сам рад је организован на следећи начин. Друга глава рада даје преглед и детаљно разматра основне алгоритме за конструкцију суфиксних стабала. Литература на основу које је ова глава писана је књига Д. Гусфиелд: Алгоритхмс он Стрингс, Треес анд Сеqуенцес . Трећа глава разматра неке од примена суфиксних стабала, такође на основу. Четврта глава разматра алгоритам ДЦ3 за конструкцију суфиксних низова. Наиме до 2003-ће године изградња суфиксних низова у линеарном времену је била могућа само полазећи од суфиксног стабла. Те године објављена су три различита алгоритма линеарне сложености за конструкцију суфиксног низа. Пета глава садржи приказ неких од примена суфиксних низова на основу рада .

Шта је суфиксно стабло

Суфиксно стабло представља структру података која описује интерну структуру стринга (ниске) на врло исцрпан начин. Суфиксно стабло може да се употреби да би се решио проблем тачног тражења за линерано време (постижући исту сложеност у најгорем случају коју су постигли алгоритми КМП (Кнутх-Моррис-Пратт) и Бојер-Мур (Боyер-Мооре), али њихова предност је у могућности примене у алгоритмима линеарне сложености за проблеме са стринговима сложенијим од тачног тражења. Штавише, суфиксна стабла обезбеђују мост између проблема тачног тражења и приближног тражења.

Дефиниција: Проблем тачног тражења је проблем налажења свих појављивања скупа стрингова P у тексту T, где је улаз целокупан скуп P.

Дефиниција: Проблем приближног тражења је проблем тражења поклапања стрингова из скупа P у тексту T, при чему су дозвољене извесне ограничене различитости у виду замена, уметања и брисања а где је улаз целокупан скуп.

Дефиниција: Проблем речника је специјални случај претраживања скупова стрингова (који заједно чине речник) чији је задатак да пронађе задати текст у речнику.

Класичан пример примене суфиксних стабала је проблем подстринга: Дат је (дугачак) текст T дужине m . Након О(m) , односно после линерног времена предобраде, морамо да будемо спремни да за сваки учитани стринг S дужине n за време О(n) пронађемо појаву S у T или да установимо да S није садржан у T. То значи да дозвољена предобрада траје пропроционално дужини текста, али након тога претраживање S мора да се уради у времену пропорционалном дужини S, независно од дужине T. Ова сложеност се достиже помоћу суфиксног стабла. Суфиксно стабло се у фази предобраде изграђује за време О(m). Након тога кад год добијемо стринг дужине О(n) алгоритам га проналази у времену О(n) користећи суфиксно стабло.

Realizacija:

Уконенов алгоритам конструише имплицитно суфиксно стабло Ii за сваки префикс S[1..n] за , почевши од I1 увећава i за један док стабло Im не буде конструисано.

Право суфиксно стабло за S је конструисано из Im а време потрошено за цео алгоритам је О(m).

Уконенов алгоритам ће бити објашњен најпре презентовањем метода у времену О(m^3) да би се kонструисала сва стабла Ii а затим да би се оптимизовањем његове имплементације добио споменути временски ниво сложености.

Уконенов алгоритам је подељен у m фаза. У фази i+1, стабло Ii+1 је конструисано из Ii. Свака фаза i+1 је даље подељена у i+1 продужења, по једно за сваки од i+1 суфикса за S[1..i+1]. У продужењу j у фази i+1, алгоритам најпре налази крај пута од корена означеног подстрингом s[j..i]. Тада алгоритам продужава подстринг додајући знак s(i+1) на његове крајеве, све дотле док се S(i+1) већ не појави тамо. Тако да је у фази i+1, стринг

S[1..i+1] први стављен у стабло, праћен стринговима S[2..i+1],S[3..i+1] , ...

Дефиниција: Суфикс S[i..j] је празан суфикс ако је i>j

For i=1 to m-1

begin {фаза i+1}

for j=1 to i+1

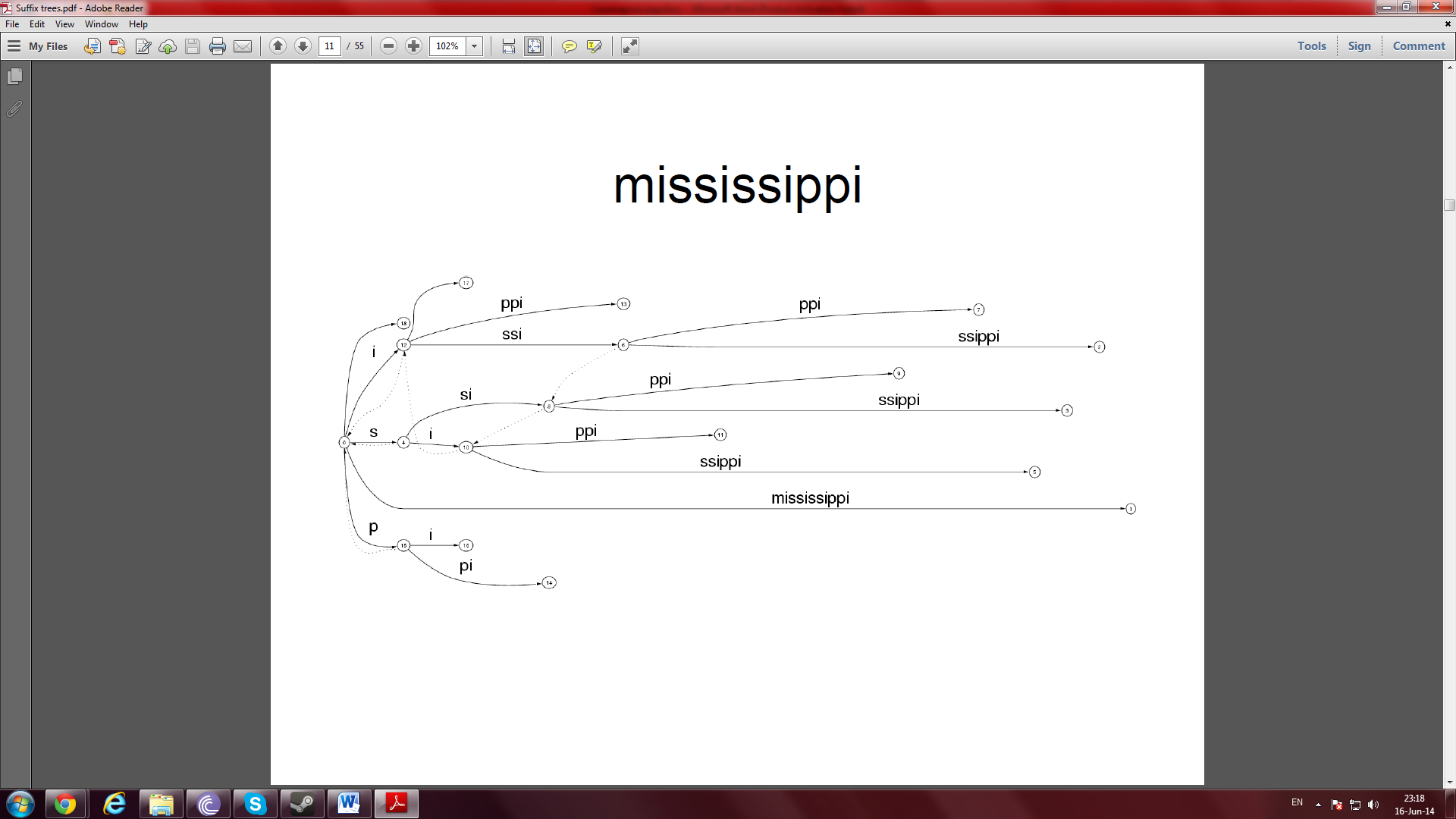
begin {продузење j}

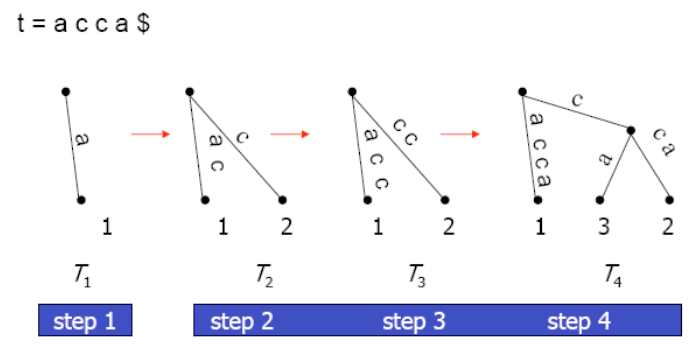
Полазећи од корена налази се крај пута означеног са S[j..i] у текућем стаблу. Ако је потребно, тај пут се продужава додавањем знака S(i+1), и тиме осигурава да је стринг S[j..i+1] у стаблу.

end;

end;

Primer:





Pravila za produženje sufiksa

Da bi se transformisao ovaj opis višeg nivoa u algoritam, mora tačno da se odredi kako da se predstavi sufiksno produženje. Pustimo S[j..i]=β da bude sufiks od S[j..i]. U produženju j, kada algoritam naђe kraj β u tekućem stablu, ono produžuje β da bi se osiguralo da je sufiks β S(i+1) u stablu. Ono radi po narednom od sledeća tri pravila:

Pravilo 1: U tekućem stablu, put β se završava u listu. To znači da se put od korena označenog sa β, produžava do kraja neke grane sa listom. Da bismo unapredili ovo stablo, znak S(i+1) se dodaje na kraj oznake na toj grani sa listom.

Pravilo 2: Ni jedan put od kraja stringa β ne počinje znakom S(i+1), ali barem jedan označen put nastavlja od kraja β.

U ovom slučaju, nova grana sa listom koja počinje od kraja β mora biti napravljena i označena znakom S(i+1). Novi čvor će takoђe morati tamo da bude napravljen ako se β završava unutar grane. Listu na kraju nove grane sa listom je dodeljen broj j.

Правило 3: Неки пут од краја стринга β почиње знаком S(i+1). У овом случају стринг β S(i+1) је већ у текућем стаблу, тако да се даље не ради ништа (сетимо се да у имплицитном суфиксном стаблу крај суфикса не мора да буде експлицитно означен).

Убрзавање алгоритма:

Користећи правила за продужење суфикса, једном када је крај суфикса β S(i+1) пронађен у текућем стаблу, потребно је само константно време да би се извршила правила за продужење суфикса (да би се осигурало да се суфикс β S(i+1) налази у стаблу). Кључна ставка у имплементацији Уконеновог алгоритма је заправо како пронаћи крајеве свих i+1 суфикса за S[1..i].

Идеја је пронаћи крај сваког од i+1 суфикса β у времену О(|β|) идући од корена текућег стабла. Овим приступом, продужење ј у фази i+1 би трајало О(i+1-j) времена, Ii+1 би могао да се направи из Ii у времену О(i^2) док би Im могао да се направи у времену О(m^3) . Овај алгоритам изгледа мало будаласто с обзиром на то да је већ познат директан алгоритам за изградњу суфиксног стабла у времену О(m^2) , али је лакше описати Уконенов О(m) алгоритам као убрзање за горе споменути метод О(m^3).

Горе споменути О(m^3) ниво сложености може да се редукује до времена О(m) са неколико запажања и трикова имплементације. Сваки трик по себи изгледа као сензибилни хеуристички метод убрзања алгоритма, али делујући појединачно ови трикови не морају обавезно да редукују ниво сложености најгорег случаја. Међутим заједно, они достижу ниво сложености најгорег случаја. Најважнији елемент убрзања је кориштење суфиксних линкова.

Суфикс Линкови

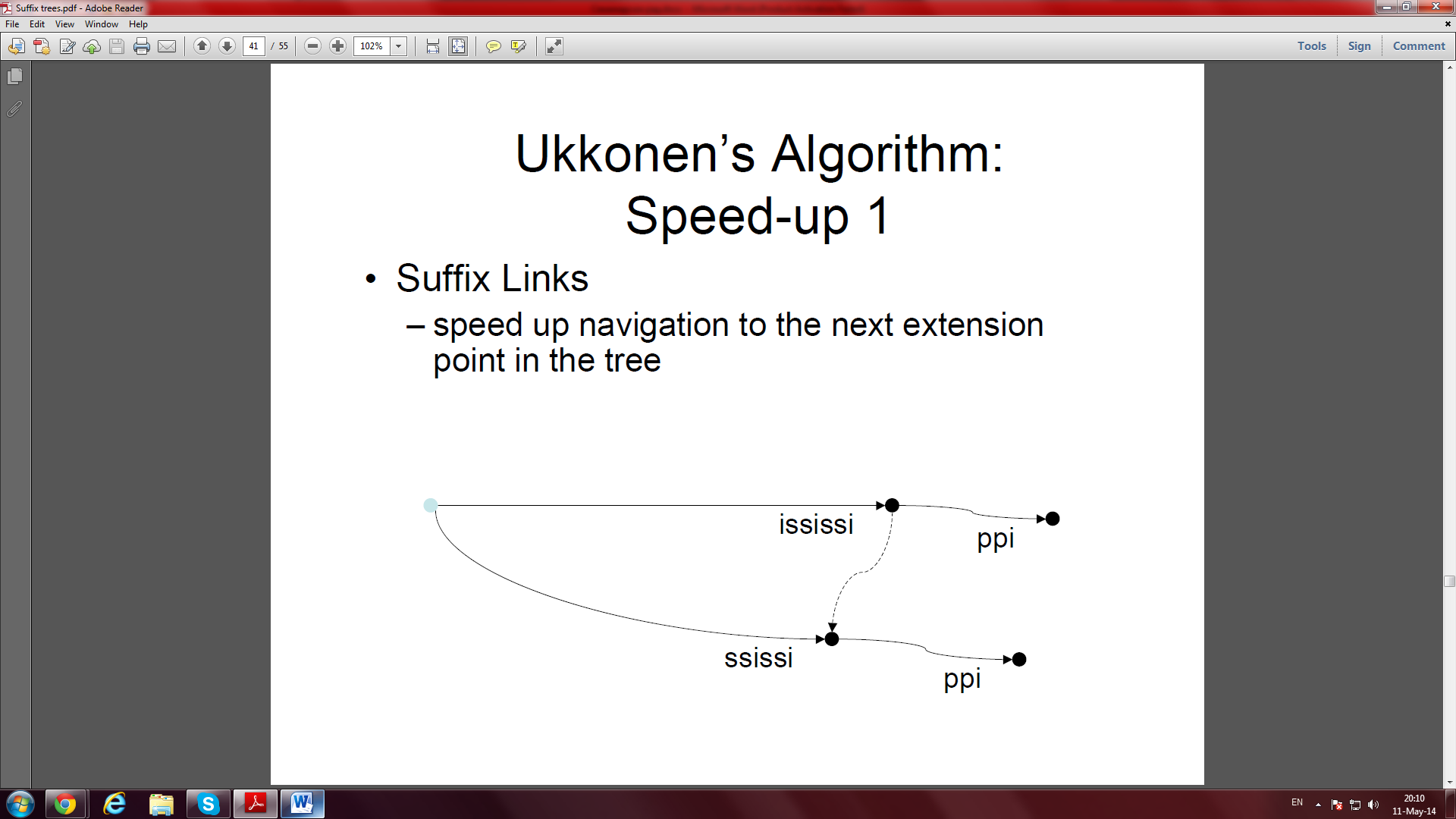
Дефиниција: Нека је са xα означен произвољни стринг, где x означава појединачни знак а α означава (могуће празан) подстринг. За унутрашњи чвор v са ознаком пута xα, ако је тамо други чвор s(v) са ознаком пута, α тада се показивач од s(v) до назива суфиксни линк.

Ако је додат нови унутрашњи чвор v са ознаком пута xα текућем стаблу у продужењу j неке фазе i+1 , тада се пут означен са α завршава већ на унутрашњем чвору текућег стабла или ће унутрашњи чвор на крају стринга да се направи (помоћу правила о продужењу) у продужењу j+1 у истој тој i+1 фази .

Posledice:

-У Уконеновом алгоритму, сваки новонаправљени унутрашњи чвор ће имати суфиксни линк од себе до краја наредног продужења.

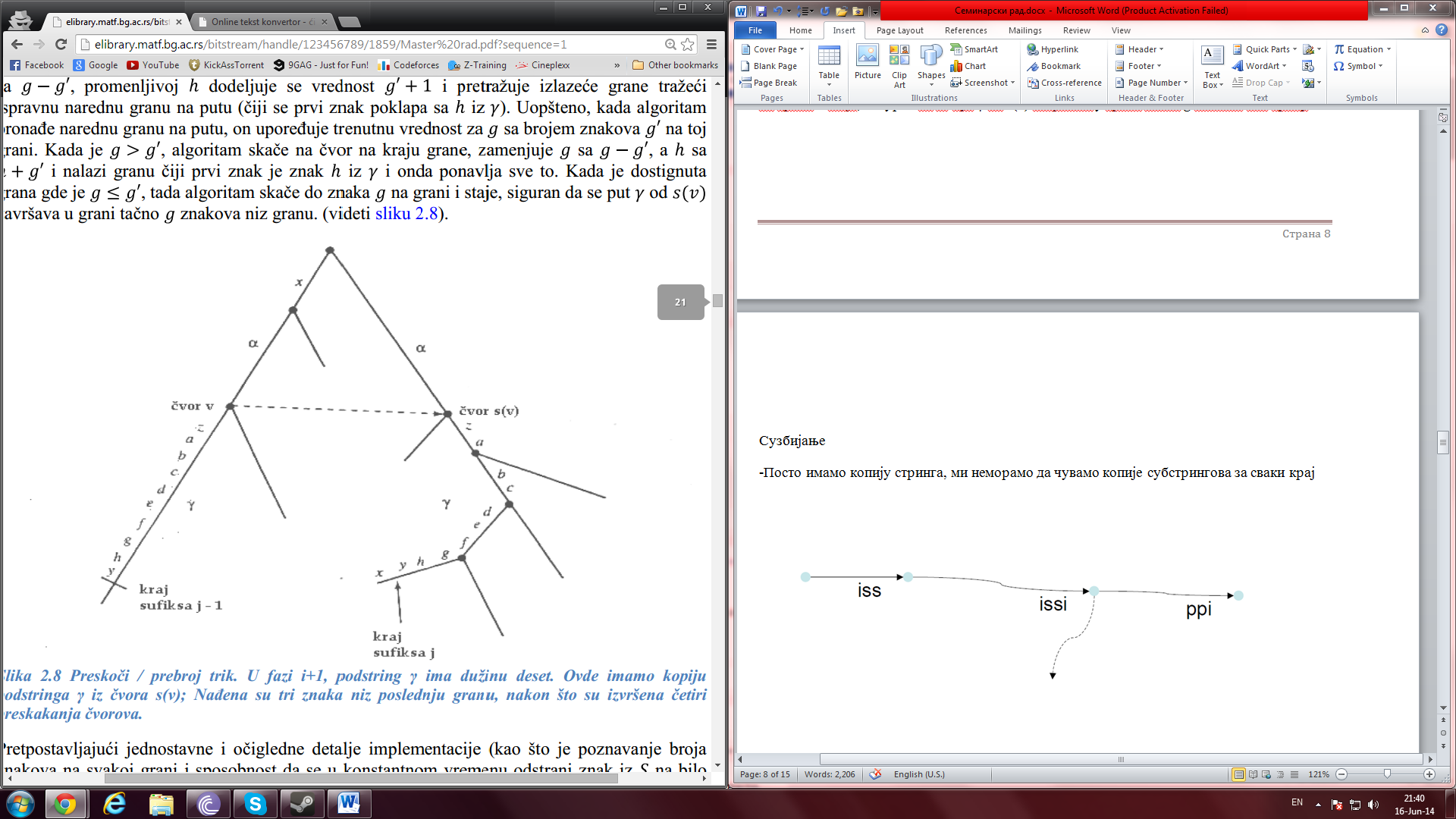
- У сваком имплицитном суфиксном стаблу Ii, ако унутрашњи чвор v има ознаку пута xα тада се тамо налази чвор S(v) из Ii са ознаком пута α.



Прескочи трик

Означимо са g дужину γ и присетимо се да никоје две ознаке грана из s(v) не могу да почну истим знаком, тако да први знак γ мора да се појави као први знак на тачно једној грани;

Нека g’ представља број знакова на тој грани. Ако g’ је мање од g тада алгоритам више не мора даље да разматра знаке. Једноставно прескаче до чвора на крају гране. Вредност g замењује се g-g’ са , променљивој h додељује се вредност g’+1 и претражује излазеће гране тражећи исправну наредну грану на путу (чији се први знак поклапа h са из γ). Уопштено, када алгоритам пронађе наредну грану на путу, он упоређује тренутну вредност за g са бројем знакова g’ на тој грани. Када је g>g’, алгоритам скаче на чвор на крају гране, замењује g са g-g’, а h са h+g’ и налази грану чији први знак је знак h из γ и онда понавља све то. Када је достигнута g<=g’ грана где је , тада алгоритам скаче до знака на грани и стаје, сигуран да се пут γ од s(v) завршава у грани тачно g знакова низ грану.



Нека (v,s(v)) је било који суфиксни линк постављен за време Уконеновог алгоритма. У том моменту, дубина чвора за v је за највише један већа од дубине чвора за s(v).

Користећи прескочи / преброј трик, било која фаза Уконеновог алгоритма узима О(m) времена.

Алгоритам за појединачно продужење

1. Проналази се први чвор v у или изнад краја S[j-1..i] који или има суфиксни линк или је корен. Ово захтева ход највише за једну грану од краја S[j-1..i] у текућем стаблу. Означимо са γ (могуће празан) стринг између v и краја S[j-1..i] .

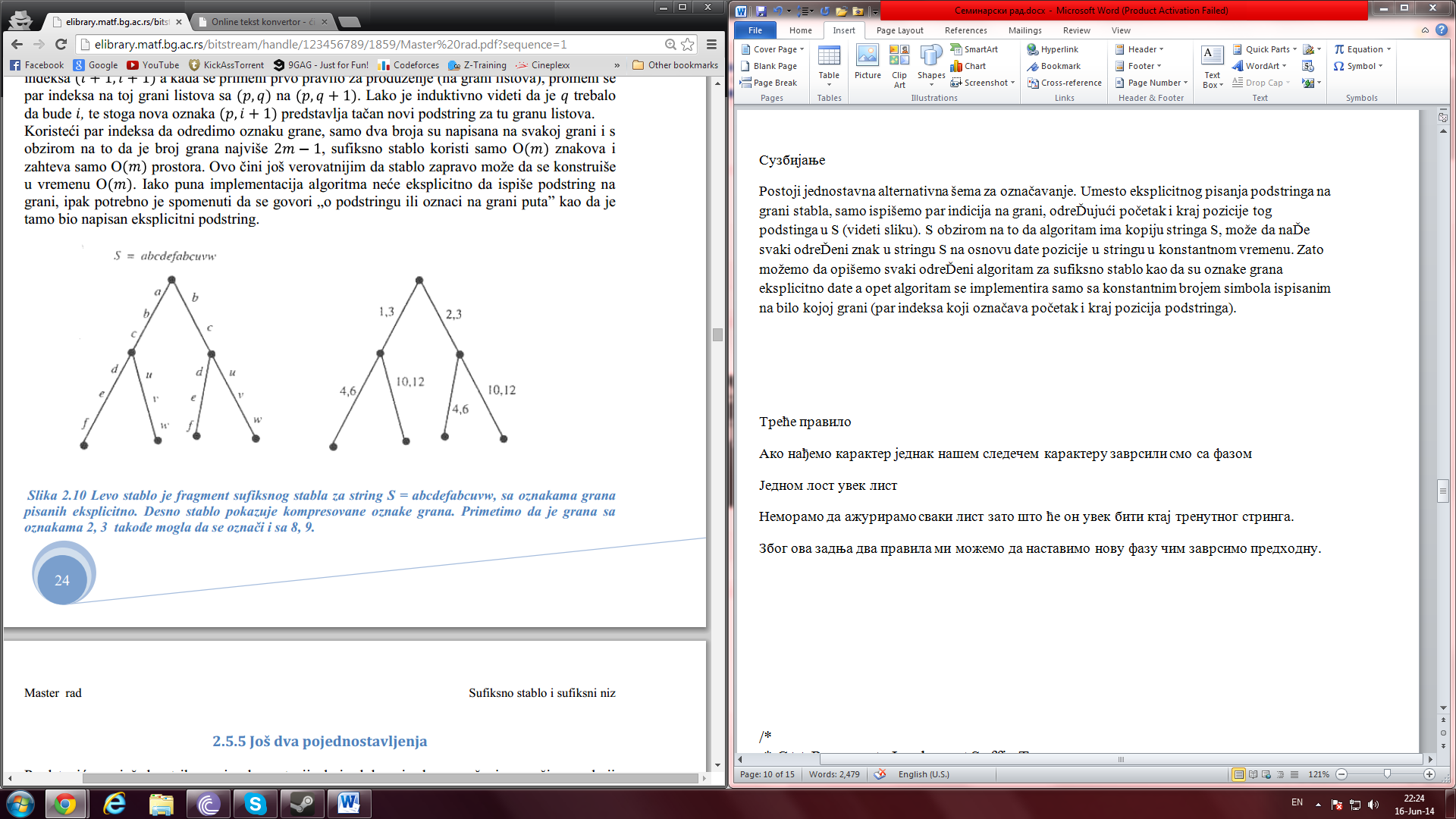
2. Ако v није корен, обилази се суфиксни линк од v до s(v) и онда се иде од s(v) пратећи пут за стринг γ. Ако је v корен, тада се прати пут за S[j..i] од корена (као у једноставном алгоритму).

3. Коришћењем правила за продужење, осигура се да је стринг S[j..i]S(i+1) у стаблу.

4. Ако је направљен нови унутрашњи чвор w у продужењу j-1 (по другом правилу за продужење), тада стринг α мора да се заврши у чвору s(w), завршном чвору за суфиксни линк из w. Сада се направи суфиксни линк (w,s(w)) од w до s(w) .

Сузбијање

Postoji jednostavna alternativna šema za označavanje. Umesto eksplicitnog pisanja podstringa na grani stabla, samo ispišemo par indicija na grani, odreђujući početak i kraj pozicije tog podstinga u S (videti sliku). S obzirom na to da algoritam ima kopiju stringa S, može da naђe svaki odreђeni znak u stringu S na osnovu date pozicije u stringu u konstantnom vremenu. Zato možemo da opišemo svaki odreђeni algoritam za sufiksno stablo kao da su oznake grana eksplicitno date a opet algoritam se implementira samo sa konstantnim brojem simbola ispisanim na bilo kojoj grani (par indeksa koji označava početak i kraj pozicija podstringa).



Правило 3 је заустављач процеса

Ако нађемо карактер једнак нашем следечем карактеру заврсили смо са фазом.

Зауставља сваку фазу i+1 први пут када је примењено треће правило о продужењу. Ако се ово догоди у продужењу j, тада више нема потребе да се експлицитно налази крај сваког стринга S[k..i] за k>j. За продужења у фази i+1 која су “завршена” након првог извршења трећег правила о продужењу се каже да су урађена имплицитно. Ово је супротно у односу на свако продужење ј где је крај за S[j..i] нађен експлицитно. Продужење те врсте се назива експлицитно продужење.

Једном лост увек лист

Односно, ако је у некој тачки у Уконеновом алгоритму направљен лист који је означен са j, (за суфикс који почиње на позицији j у S) тада ће лист остати у свим стаблима наследницима направљеним током алгоритма. Ово је истина, јер алгоритам нема механизме за продужење гране листова даље од његовог тренутног листа. Детаљније, када је једном лист означен са j, прво правило за продужење ће увек да се примени до продужења j у свакој наредној фази. Тако да, једном лист - заувек лист.

Због ова задња два правила ми можемо да наставимо нову фазу чим заврсимо предходну.

/\*

\* C++ Program to Implement Suffix Tree

\*/

#include <iostream>

#include <cstdlib>

#include <cstring>

#include <string>

using namespace std;

typedef unsigned char byte;

/\*

\* SuffixNode Declaration

\*/

class SuffixNode

{

public:

int depth, begin, end;

SuffixNode \*\*children;

SuffixNode \*parent, \*suffixLink;

/\*

\* Constructor

\*/

SuffixNode(int begin, int end, int depth, SuffixNode \*parent)

{

children = new SuffixNode\* [38];

this->begin = begin;

this->end = end;

this->parent = parent;

this->depth = depth;

}

bool contains(int d)

{

return depth <= d && d < depth + (end - begin);

}

};

string alphabet;

int alphabetSize;

int lcsLength;

int lcsBeginIndex;

/\*

\* Class SuffixTree Declaration

\*/

class SuffixTree

{

public:

/\*

\* Funtion to build suffix tree for given text

\*/

SuffixNode \*buildSuffixTree(string s)

{

int n = s.length();

int \*a = new int[n];

for (int i = 0; i < n; i++)

{

a[i] = alphabet.find(s.at(i));

}

SuffixNode \*root = new SuffixNode(0, 0, 0, NULL);

SuffixNode \*cn = root;

root->suffixLink = root;

SuffixNode \*needsSuffixLink = NULL;

int lastRule = 0;

int j = 0;

for (int i = -1; i < n - 1; i++)

{

int cur = a[i + 1];

for (; j <= i + 1; j++)

{

int curDepth = i + 1 - j;

if (lastRule != 3)

{

if (cn->suffixLink != NULL)

cn = cn->suffixLink;

else

cn = cn->parent->suffixLink;

int k = j + cn->depth;

while (curDepth > 0 && !cn->contains(curDepth - 1))

{

k += cn->end - cn->begin;

cn = cn->children[a[k]];

}

}

if (!cn->contains(curDepth))

{

if (needsSuffixLink != NULL)

{

needsSuffixLink->suffixLink = cn;

needsSuffixLink = NULL;

}

if (cn->children[cur] == NULL)

{

cn->children[cur] = new SuffixNode(i + 1, n, curDepth, cn);

lastRule = 2;

}

else

{

cn = cn->children[cur];

lastRule = 3;

break;

}

}

else

{

int end = cn->begin + curDepth - cn->depth;

if (a[end] != cur)

{

SuffixNode \*newn = new SuffixNode(cn->begin, end, cn->depth, cn->parent);

newn->children[cur] = new SuffixNode(i + 1, n, curDepth, newn);

newn->children[a[end]] = cn;

cn->parent->children[a[cn->begin]] = newn;

if (needsSuffixLink != NULL)

needsSuffixLink->suffixLink = newn;

cn->begin = end;

cn->depth = curDepth;

cn->parent = newn;

cn = needsSuffixLink = newn;

lastRule = 2;

}

else if (cn->end != n || (cn->begin - cn->depth) < j)

{

lastRule = 3;

break;

}

else

lastRule = 1;

}

}

}

root->suffixLink = NULL;

return root;

}

/\*

\* Funtion to find longest common substring

\*/

int findLCS(SuffixNode \*node, int i1, int i2)

{

if (node->begin <= i1 && i1 < node->end)

return 1;

if (node->begin <= i2 && i2 < node->end)

return 2;

int mask = 0;

for (int f = 0; f < alphabetSize; f++)

{

if (node->children[f] != NULL)

{

mask |= findLCS(node->children[f], i1, i2);

}

}

if (mask == 3)

{

int curLength = node->depth + node->end - node->begin;

if (lcsLength < curLength)

{

lcsLength = curLength;

lcsBeginIndex = node->begin;

}

}

return mask;

}

/\*

\* Funtion to find longest common substring

\*/

void findLCS(string s1, string s2)

{

string x = "-";

string y = "#";

string ns1 = s1;

string ns2 = s2;

string s = s1.append(x.append(s2.append(y)));

SuffixNode \*root = buildSuffixTree(s);

lcsLength = 0;

lcsBeginIndex = 0;

findLCS(root, ns1.length(), ns1.length() + ns2.length() + 1);

bool chk = lcsLength > 0;

if (chk)

{

cout<<"\nLongest substring is "<<s.substr(lcsBeginIndex , lcsLength);

cout<<endl;

}

else

{

cout<<"No substring found"<<endl;

}

}

};

/\*

\* Main

\*/

int main()

{

alphabet = "abcdefghijklmnopqrstuvwxyz1234567890-#";

alphabetSize = alphabet.length();

string s1,s2;

cout<<"Finding longest common substring using suffix trees\n";

cout<<"Enter 1st String: ";

cin>>s1;

cout<<"Enter 2nd String: ";

cin>>s2;

SuffixTree st;

st.findLCS(s1, s2);

return 0;

}