ФЕДЕРАЛЬНОЕ АГЕНТСТВО ЖЕЛЕЗНОДОРОЖНОГО ТРАНСПОРТА

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Иркутский государственный университет путей сообщения

(ФГБОУ ВО ИрГУПС)»

Факультет:

Кафедра: Информационные системы и защита информации

Реализация алгоритма построение LCS – самой длинной общей подпоследовательности двух строк (Алгоритм Нидлмана- Вунша)

Курсовая работа

КР.430200.15.03.06.2016.П3

Выполнил: Проверил:

Студент группы МР-14-1 ст. преподаватель

М. В. Баканов В. А. Лучников

«\_\_\_»\_\_\_\_\_\_\_2016г. «\_\_\_»\_\_\_\_\_\_\_2016г.

Иркутск 2016Аннотация

Целью курсовой работы является изучение и программная реализация алгоритма поиска самой длинной общей подпоследовательности двух строк (LCS – Longest Common Subsequence)

В работе рассмотрен алгоритм Нидлмана- Вунша. Этот алгоритм является примером динамического программирования и стал первым алгоритмом, примененным для сравнения биологических последовательностей.

Содержание

**Введение4**

**Формализация задачи5**

**Решение задачи методом динамического программирования**6

**Алгоритм Нидлмана- Вунша9**

**Программная реализация задачи построения LCS на языке СИ12**

**Заключение13**

**Список использованной литературы14**

**Приложение А – Блок- схемы программы15**

**Приложение Б – Код программы17**

**Приложение В – Проверка работы программы19**

Введение

Задача о нахождении LCS нескольких строк имеет приложения в задаче сравнения текстовых файлов и биоинформатике. Для решения данной задачи разработано много различных алгоритмов, отличающихся друг от друга сложностью. Например, алгоритм Хиршберга, Вагнера- Фишера, Ханта- Шиманского.

Время выполнения алгоритма LCS достаточно легко рассчитывается, поскольку он управляется двумя вложенными циклами: внешний выполнят n(эн) итераций в секунду, а внутренний – m(эм).

В работе рассмотрен алгоритм, обладающий меньшей временной сложностью и линейной сложностью по памяти.

Подпоследовательность можно получить из некоторой конечной строки, если удалить из неё множество символов (возможно и пустое).

Общая подпоследовательность двух строк – это строка, являющаяся подпоследовательностью каждой из них. Самая длинная из таких строк – наибольшая общая подпоследовательность(LCS). Таких подпоследовательностей может быть несколько.

Задачи курсовой работы:

1. Изучение алгоритма LCS
2. Программная реализация
3. Оценка эффективности

«1» Формализация задачи

По-другому условие задачи можно записать так: имеются два файла с текстами. Требуется выяснить, в какой мере эти тексты идентичны.

При анализе поставленной проблемы был отмечен тот факт, что задача определения степени идентичности двух текстовых файлов может быть сведена к задаче поиска максимальной общей подпоследовательности для двух последовательностей.

Прямо сравнивать два при этом файла не рационально. В частности, можно добавить в один из файлов пустые строки, пробелы, комментарии, и тогда процент совпадения станет намного меньшим, кроме того, смена регистра символов также окажет влияние повлиять на степень совпадения. Но ведь тексты могут полностью совпадать. Решение данной проблемы заключается в том, что перед поиском максимальной подпоследовательности необходимо определенным образом преобразовать исходные файлы.

Так, если из текста удалить пробелы, комментарии, переводы строк и привести все символы к одному регистру (проще всего к верхнему) то это позволит более точно определить степень совпадения. Сравнение преобразованных таким образом файлов даст результаты, гораздо более близкие к искомой величине.

Следовательно, поставленную задачу можно разбить на две подзадачи:

1. Нахождение максимальной подпоследовательности для двух полученных последовательностей.
2. Преобразование текста программы в последовательность символов в верхнем регистре без пробелов, символов перевода строки, комментариев.
3. Существует несколько разных способов определения подобия двух строк. Один из наиболее простых и часто встречающихся способов использует символьные строки и их подпоследовательности. **Определение:** Подпоследовательностью строки *X* будет любая строка, имеющая вид *xixi2...xik*, где *ij < ij + 1*; т.е. это набор символов, необязательно расположенных подряд, но, тем не менее, существующих в *Х*.

Например, строка ***AAAG*** является подпоследовательностью строки ***CGATAATTGAGA***. (Понятие подпоследовательности строки отличается от определения подстроки. В подстроке все символы обязательно расположены подряд).

Задача нахождения наиболее длинной общей подпоследовательности известна в англоязычной литературе под названием **Longest Common Subsequence (LCS)**. Она состоит в том, чтобы из двух строк символов некоторого алфавита определить наиболее длинную строку *S*, являющуюся наиболее длинной подпоследовательностью обеих строк.

Одним из вариантов решения этой задачи является перебор всех подпоследовательностей в *Х* и выбор самой длинной из них, одновременно являющейся подпоследовательностью в *Y*. Поскольку каждый символ в X либо находится в подпоследовательности, либо нет, существует потенциально *2n*различных подпоследовательностей *Х,* каждая из которых требует *О(m)* времени для определения, является ли она подпоследовательностью в *Y.* Такой метод не эффективен.

* 1. *Решение задачи с помощью динамического программирования*

Методика динамического программирования используется для оптимизационных задач, когда требуется определить наилучший способ выполнения некоторого действия. Зачастую количество разных способов выполнения такого действия экспоненциально, поэтому примитивный поиск недопустим. Если задача содержит определенное количество структур, которые могут быть использованы. Такая структура включает три следующих компонента.

**Простая подзадача:** общая оптимизированная задача должна иметь возможность разбиения на подзадачи. Более того, должен существовать простой способ обозначения подзадач всего несколькими индексами типа *i, j, k* и т.п.

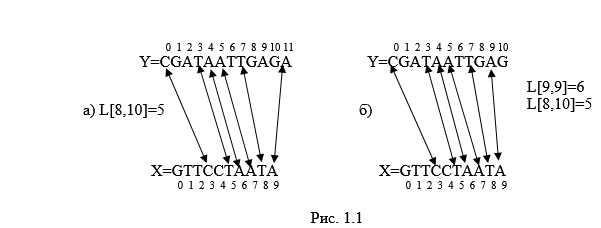
**Оптимизация подзадачи:** оптимальным решением общей задачи должна стать комбинация оптимальных решений подзадач. Вряд ли возможно найти оптимальное решение всей задачи, содержащее оптимальные подзадачи.

**Наложение подзадач:** оптимальные решения не связанных между собой подзадач могут сами содержать общие подзадачи.

Рассмотрим применение технологий динамического программирования для поиска максимальной подпоследовательности.

Для решения задачи имеется две строки *X* и *Y*. Т.к. *X* и *Y* состоят из символов, имеется естественное множество индексов, с помощью которых можно определить подзадачи – индексы в строке *X* и в строке *Y*.

Определим подзадачи таким образом, чтобы использовать при вычислениях значения *L[i,j]*, применяемого для хранения длины наиболее длинной строки, являющейся подпоследовательностью обеих *X = x1x2x3x4...xn-1* и*Y = y1y2y3y4...ym-1*. Это позволит переопределить *L[i,j]*для оптимального решения подзадачи, что зависит от ситуации, которая может быть следующей (см. рис. 1.1).



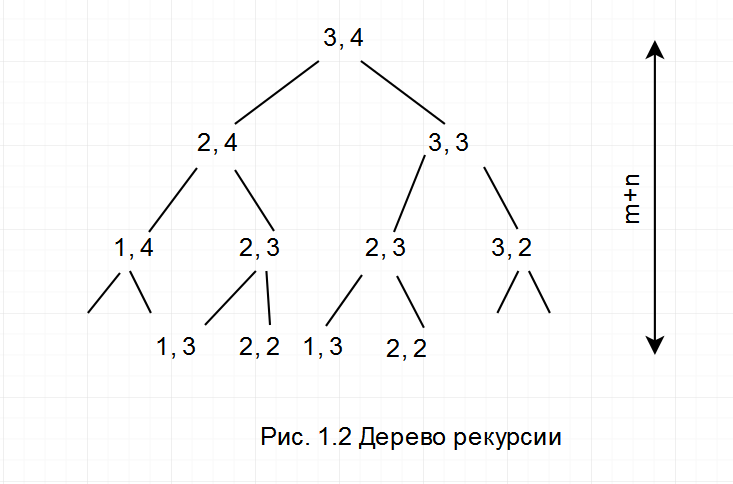
При работе алгоритма возможны два случая наиболее длинной общей подпоследовательности: а) *xi=yj* и б) *xi≠yj*. Заметим, что алгоритм сохраняет только значения *L[i,j]*, а не совпадения.

Пусть L[I,j]=LCS(x[1..i],y[1..j]) – длина общей подпоследовательности для префиксов x и y, длины i и j соответственно.Вычислив для всех i и j, получим L[m, n]= LCS(x,y) – искомая наибольшая подпоследовательность.

Первый признак динамического программирования: оптимальное решение исходной задачи содержит оптимальные решения подзадач меньшего размера. В случаи задачи LCS это означает, что если z = LCS(x, у), то любой префикс z является LCS префикса х и префикса у.

Рассмотрим случай x[i] = y[i]. Пусть z[l .. k] = LCS(x[l .. i], y[l .. j])— LCS префиксов, тогда k — длина z. Тогда последний символ подпоследовательности должен быть равен z[k] = x[i](= y[j]) иначе z можно было бы удлинить, добавив x[i].

В худшем случае, если x[i]= y[j] алгоритм выполняет рекурсивное вычисление двух подзадач, по очереди декрементируя один из параметров. Дерево рекурсии алгоритма имеет высоту m+n (Рис.1.2), т.е. время выполнения по-прежнему будет экспоненциальным.



Можно заметить, что некоторые поддеревья в нём повторяются несколько раз. Вычисляя соответствующие подзадачи однократно, время работы можно сократить.

Это свойство называется "перекрывающимися подзадачами" или вторым признаком динамического программирования.

Второй признак динамического программирования: рекурсивное решение содержит небольшое количество различных подзадач, которые повторяются многократно. Количество различных подзадач в LCS равно п.т, что гораздо меньше n.2m.

«2» Алгоритм Нидлмана- Вунша

Алгоритм Нидлмана — Вунша — это очень простой и понятный алгоритм, основанный на двумерном ДП, предназначенный для выполнения выравнивания двух последовательностей, который используется в биоинформатике при построении выравниваний аминокислотных или нуклеотидных последовательностей. Алгоритм был предложен в 1970 году Солом Нидлманом и Кристианом Вуншем.

Превращение описанного определения L[ij] в алгоритм представляется достаточно простым и наглядным.

Создается массив размера [-1..п+1,-1..т+1], L для пограничных случаев, когда i=0 и j=0. То есть L[i,-1]=0, при /=0,7, ...,«-7 и L[-l,j]=0, при j=0,l, ...,m-7 (при этом порядок несколько нарушается, поскольку сроки и столбцы в L должны индексироваться с О). Затем итеративно определяем значения L до тех пор, пока не получим L[n-l, т-1], длину наибольшей общей подпоследовательности X и Y

**Пример:** Рассмотрим две строчки: A = “bedcadb” и B = “abcdede”.    
Для начала заполним матрицу L по этим 4 принципам:

1) Буквы строчки A напишем перед строками матрицы, а элементы строчки B над столбцами матрицы.   
2) Матрица будет иметь нулевые столбец и строку, состоящие из нулей.   
3) Саму матрицу давайте назовем L.   
4) В элементе матрицы L[i][j] будет хранится длина наибольшей общей подпоследовательности для префиксов A[1..i] и B[1..j].

Из п.4 делаем вывод, что нумерация букв в строках с единицы.   
Префикс A[1..i] – это первые i символов строки A.

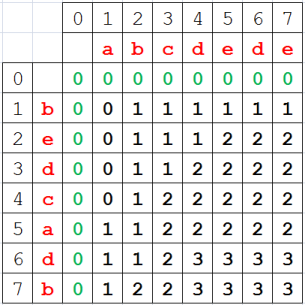
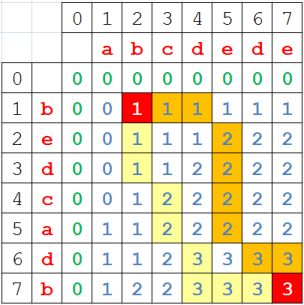
 

Рис. 2.1 Рис. 2.2

Разберем элемент L[3][4]. В данном случае рассматриваются два префикса: “bed” и “abcd”. Чисто интуитивно можно догадаться, что lcs для этих двух строк будет “bd”, поэтому в самом элементе храним 2. Все остальные элементы можно заполнить интуитивно.   
Рекуррентные формулы:   
                
**| L[i-1][j-1] + 1, если a[i] == b[j]   
L[i][j] = <    
           | max(L[i-1][j],L[i][j-1]), иначе**Т.Е. длина общей последовательности двух префиксов на единицу больше предыдущих префиксов, если их конечные элементы совпадают, или равны максимуму из двух длин для двух предыдущих префиксов в противном случае.

Результирующая длина lcs для полных строк a и b находится в элементе L[7][7]. **Оранжевая lcs: bed; Желтая lcs: bcd.**

«3» Программная реализация задачи построения LCS на языке СИ

Существуют четыре функции. Все они вызывают и используют результат друг друга, каждая в своих целях.

Головная программа (см. приложение А-1) считывает строки и вставляет их в функцию output lсs Need Vunsh. Она считает, сколько в этих строках символов (strlen(a),strlen(b)).

Далее - два вложенных цикла. Эти два цикла как раз и формируют матрицу

matr[ ][ ]. Функция memset() просто заполняет нулями матрицу matr[ ][ ]. Далее мы сравниваем строки. Если находится совпадение — соседний по ходу элемент матрицы плюсуется на единицу. Если не находится — берётся максимальный из соседнего (функция mах, см. приложение А-4).

Как только матрица заполнится — выводится длина LCS (она автоматически получатся в нижнем правом элементе matr[n][m]).

Теперь, когда имеется длина LCS и заполненная матрица, остается получить LCS. Обратным ходом восстанавливаем её. Эта функция вызывается out ans(n,m) (см. см. приложение А-З). Исходные данные для неё — это длина и ширина матрицы. Цикл while обходит матрицу наоборот — с последнего элемента mem[n][m]. Вынимать саму LCS будем в стек.

Параллельно строки снова сравниваются. Если символы в строках совпали — их позиция записывается в стек. Если же не совпали, то смотрится матрица. Если элемент справа сверху больше текущего (с которым сравниваем), то делается шаг вверх, иначе — шаг влево.

После того, как этот цикл закончится, в стеке получим номера позиций в строке символов искомой LCS. Она выводится вторым циклом while.

«Верхушка» стека (функция pop) копирует и выводит символ строки с позиции этой самой «верхушки». После этого, стек сдвигается (функция pop — она удаляется и нижние элементы сдвигаются вверх). И заново. В итоге выводится LCS.

Заключение

В курсовой работе исследована возможность использования алгоритма поиска максимальной подпоследовательности (*Longest Common Subsequence*) для оценки степени совпадения тестов в файле. Также было проведено подробное изучение и программная реализация самого алгоритма LCS. Довольно тщательно проведена оценка быстродействия не только теоретически, но и практически. При определении быстродействия были учтены как лучший, так и худший случай.

Но есть у данной программы недостатки. Довольно легко изменить файл с текстом так, чтобы степень совпадения значительно снизилась. Например, можно изменить названия разделов н схожие по смыслу, изменить структуру предложений. После такой операции, если текст небольшой, степень совпадения упадет значительно. Стоит также заметить, что данную процедуру добавить в программу, без падения ее производительности, довольно сложно.

В дальнейшем планируется усовершенствовать программу таким образом, что выше перечисленные действия не смогут повлиять на степень совпадения текстов.

Список использованной литературы

1. Лучников В.А. Программирование на языке СИ. Учебное пособие Иркутск: ИрГУПС, 2014. – 157 с.
2. Громов Ю.Ю., Татаренко С.И. Программирование на языке СИ: Учебное пособие. – Тамбов, 1995 – 169с.
3. Билл Смит. Методы и алгоритмы вычислений на строках. Теоретические основы регулярных вычислений: Книга – Москва: Вильямс, 2006-496с.
4. Панин А.Г. Об одном алгоритме поиска наибольшей общей подпоследовательности двух строк. Научная статья – Тольятти: ТГУ, 2010 – 3с.
5. Гасфилд Д. Строки, деревья и последовательности в алгоритмах. Информатика и вычислительная биология. Книга – Санкт- Петербург? Невский Диалект, 2003 – 248с.

Приложение А – Блок- схемы программы

1. Головная программа

a,b-симв, matr[ ][ ]- целый

output\_lcs\_Need\_Vunsh(a,strlen(a),b,strlen(b)

Ввод строк a и b

Начало

Останов

1. Процедура output\_lcs\_Need\_Vunsh – заполнение матрицы

output\_lcs\_Need\_Vunsh(char \*a, char \*b, int n, int m)

matr[i][j]=matr[i-1][j-1]+1

matr[i][j]=max(matr[i][j-1], matr[i-1][j])

a[i-1] = b[j-1]

j=1, n

Заполнение массива

i=1, n

Out\_ans(n,m)

1. Процедура out\_ans

matr[i][j]=true

pos=0, n, m– целые

Стек lcsPos - целый

out\_ans(int n, int m)

ДА

a[i-1] = b[j-1]

ДА

matr[i-1][j]≥matr[i][j-1]

Положить i-1 на вершину стека lcsPos

j-1

ДА

i-1

lcsPos НЕ пуст

i-1, j-1

ДА

Верх lcsPos

Извлечение из стека

Приложение Б – Код программы

#include <string.h> // Директивы препроцессора

#include <stdio.h>

#include <stack>

#include <math.h>

using namespace std;

const int SIZE = 1e3+2;

char a[SIZE], b[SIZE];

int matr[SIZE][SIZE];

void out\_ans(int n, int m) // Эта функция достает LCS из матрицы

{

int pos = 0;

int i = n, j = m;

stack<int> lcsPos; // Создаём СТЕК

while (matr[i][j]) // Просматриваем матрицу и записываем LCS в СТЕК

{

if (a[i-1] == b[j-1])

{

lcsPos.push(i-1); // PUSH – кладём элемент на вершину стека

i--; j--;

}

else

{

if (matr[i-1][j] >= matr[i][j-1])

i--;

else

j--;

}

}

while (!lcsPos.empty())

{

printf("%c", a[lcsPos.top()]);

lcsPos.pop();

}

}

void output\_lcs\_Need\_Vunsh(char \*a, int n, char \*b, int m) // Эта функция заполняет матрицу двумя циклами

{

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

{

memset(matr[i],0,(m+1)\*sizeof(int));

for (int j=1;j<=m;j++)

{

if (a[i-1] == b[j-1])

matr[i][j] = matr[i-1][j-1] + 1;

else

{

matr[i][j] = max(matr[i][j-1],matr[i-1][j]);

}

}

}

printf("\nstring a = %s", a);

printf("\nstring b = %s\n", b);

printf("\nDlina LCS(a,b) = %d", matr[n][m]);

printf("\nLCS(a,b) = ");

out\_ans(n,m);

}

int main(void)

{

printf("\nVvedite stroku a: ");

scanf("%s", &a);

printf("\nVvedite stroku b: ");

scanf("%s", &b);

output\_lcs\_Need\_Vunsh(a,strlen(a),b,strlen(b));

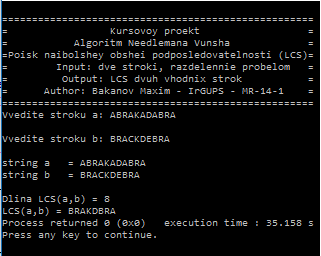
return 0;

}

Приложение В – Проверка работы программы

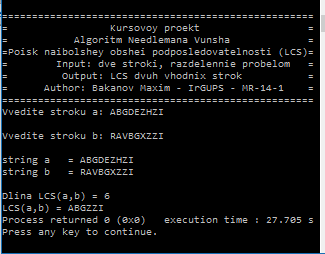
Входные данные: ABRAKADABRA/BRACKDEBRA

Выходные данные: BRAKDBRA



Входные данные: ABGDEZHZI/RAVBGXZZI

Выходные данные: ABGZZI



Входные данные: 12345/54321

Выходные данные: 1

