Московский Авиационный Институт
(Национальный Исследовательский Университет)
Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

Курсовая проект по курсу

«Дискретный анализ»

Diff

Студент: Ибрагимов Р. Р.
Группа: М8О-303Б-22
Преподаватель: Сорокин С.А.
Оценка: _____
Дата: _____
Подпись: _____

Москва, 2024

Описание задачи
Даны две строки со словами. Необходимо найти наибольшую общую подпоследовательность слов. Обратите внимание на ограничения по памяти, стандартное решение при помощи дп не подойдет. Попробуйте использовать асимптотически линейное по памяти решение.

Формат ввода
Две строки со словами разделенными пробелом.
В каждой строке не более 10^4 слов. Суммарная длина всех слов не превышает 2*10^5.

Формат вывода
В первой строке выведите одно число L — длину наибольшей общей подпоследовательности. Во второй строке через пробел выведите L слов — найденную подпоследовательность.

Теория
Наибольшая общая подпоследовательность (НОП) двух последовательностей — это максимальная по длине подпоследовательность, которая присутствует в обеих строках. Для решения задачи поиска НОП классическим способом используется алгоритм динамического программирования, который требует O(n*m) времени и памяти, где n и m — длины входных строк. Этот алгоритм основывается на построении двумерной таблицы dp, где dp[i][j] хранит длину НОП для первых i символов первой строки и первых j символов второй строки.

Процесс включает инициализацию таблицы, заполнение её значениями и последующее восстановление ответа. Вначале устанавливаются базовые значения: dp[0][j]=0 для всех j, а также dp[i][0]=0 для всех i. Затем таблица заполняется: если текущие символы строк совпадают (a[i−1]==b[j−1]), то dp[i][j]=dp[i−1][j−1]+1; иначе выбирается максимум из соседних значений: dp[i][j]=max(dp[i−1][j], dp[i][j−1]). После заполнения таблицы процесс завершается восстановлением ответа, начиная с dp[n][m] и прослеживая, какие символы входных строк вошли в НОП. Основным недостатком такого подхода является квадратичное потребление памяти, что делает его непригодным для обработки длинных строк при строгих ограничениях памяти.

Для преодоления этого ограничения используется алгоритм Хиршберга, модификация классического метода, которая снижает потребление памяти до O(n+m), сохраняя временную сложность O(n*m). Алгоритм основан на свойствах НОП, позволяющих разбивать задачу на подзадачи меньшего размера. Вместо хранения всей таблицы dp, алгоритм использует только два вектора: текущий и предыдущий столбцы. Прямой и обратный проходы по данным применяются для расчета длин НОП, что позволяет определить точку оптимального разбиения строки.

Работа алгоритма Хиршберга включает несколько шагов. Если одна из строк пуста, результатом становится пустая подпоследовательность. Если длина одной из строк равна 1, проверяется, содержится ли её слово в другой строке, и, если да, оно возвращается как НОП. В общем случае первая строка делится на две части, а вторая анализируется для поиска оптимального разбиения, при котором сумма длин НОП левой и правой частей максимальна. Затем алгоритм применяется рекурсивно к этим частям, а результаты объединяются для получения итоговой НОП.

Описание алгоритма

В программе реализованы следующие функции:

1) std::vector<std::string> split(const std::string& s) — по входной строке возвращает последовательность слов, используя в качестве разделителя пробел;
2) std::vector<int> compute_lcs(const std::vector<std::string>& a, const std::vector<std::string>& b) - вычисляет длины НОП между заданными последовательностями слов;
3) std::vector<std::string> hirschberg(const std::vector<std::string>& a, const std::vector<std::string>& b) — рекурсивный алгоритм Хиршберга (описанный в теории) для нахождения НОП между двумя последовательностями слов.

Исходный код

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

std::vector<std::string> split(const std::string& s) {
    std::vector<std::string> result;
    std::string word;

    for (char ch : s) {
        if (ch == ' ') {
            if (!word.empty()) {
                result.push_back(word);
                word.clear();
            }
        }
        else {
            word += ch;
        }
    }

    if (!word.empty()) {
        result.push_back(word);
    }
```

```
    return result;
}

std::vector<int> compute_lcs(const std::vector<std::string>& a,
const std::vector<std::string>& b) {
    size_t n = a.size(), m = b.size();
    std::vector<int> previous(m + 1, 0), current(m + 1, 0);

    for (size_t i = 1; i <= n; ++i) {
        for (size_t j = 1; j <= m; ++j) {
            if (a[i - 1] == b[j - 1]) {
                current[j] = previous[j - 1] + 1;
            }
            else {
                current[j] = std::max(previous[j], current[j -
1]);
            }
        }
        std::swap(previous, current);
    }

    return previous;
}

std::vector<std::string> hirschberg(const
std::vector<std::string>& a, const std::vector<std::string>& b) {
    size_t n = a.size(), m = b.size();

    if (n == 0 || m == 0) {
        return {};
    }
    if (n == 1) {
        for (const std::string& word : b) {
            if (word == a[0]) {
                return {a[0]};
            }
        }
        return {};
    }
    if (m == 1) {
        for (const std::string& word : a) {
            if (word == b[0]) {
                return {b[0]};
            }
        }
        return {};
    }

    size_t mid = n / 2;
```

```
    std::vector<int> lcs_lengths_left = compute_lcs({a.begin(),
a.begin() + mid}, b);
    std::vector<int> lcs_lengths_right = compute_lcs({a.rbegin(),
a.rbegin() + (n - mid)}, {b.rbegin(), b.rend()});

    size_t optimal_split = 0;
    size_t max_length = 0;

    for (size_t i = 0; i <= m; ++i) {
        size_t current_length = lcs_lengths_left[i] +
lcs_lengths_right[m - i];
        if (current_length > max_length) {
            max_length = current_length;
            optimal_split = i;
        }
    }

    std::vector<std::string> result_lcs_left =
hirschberg({a.begin(), a.begin() + mid}, {b.begin(), b.begin() +
optimal_split});
    std::vector<std::string> result_lcs_right =
hirschberg({a.begin() + mid, a.end()},{b.begin() + optimal_split,
b.end()});

    result_lcs_left.insert(result_lcs_left.end(),
result_lcs_right.begin(), result_lcs_right.end());
    return result_lcs_left;
}

int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(0);
    std::cout.tie(0);

    std::string s1, s2;

    std::getline(std::cin, s1);
    std::getline(std::cin, s2);

    std::vector<std::string> words1 = split(s1);
    std::vector<std::string> words2 = split(s2);

    std::vector<std::string> lcs = hirschberg(words1, words2);

    std::cout << lcs.size() << "\n";
    for (const std::string& word : lcs) {
        std::cout << word << " ";
    }
    std::cout << "\n";

    return 0;
}
```

Тесты

Номер	Ввод	Вывод
1	abc sdf kjb kjb kjb yu sdf kjf kl kjb	2 sdf kjb или 2 kjb kjb
2	vd y fu c n t yt wz wp e h pz r mv qm v gz a y c bs fn j yt g lp mv a	5 y c yt mv a
3	c d qwer f z xc v b a r t y a b c d e f q w e r t y	6 c d f r t y

Замеры производительности

Данные теста	Классический НОП	НОП алгоритмом Хиршберга
n = 100, m = 100 длина слов от 1 до 4	Classical LCS: Length: 4 Time taken: 1 ms Memory used: 128 KB	Hirschberg LCS: Length: 4 Time taken: 2 ms Memory used: 0 KB
n = 1000, m = 1000 длина слов от 1 до 4	Classical LCS: Length: 80 Time taken: 45 ms Memory used: 3968 KB	Hirschberg LCS: Length: 80 Time taken: 55 ms Memory used: 64 KB
n = 5000, m = 5000 длина слов от 1 до 6	Classical LCS: Length: 276 Time taken: 805 ms Memory used: 97664 KB	Hirschberg LCS: Length: 276 Time taken: 1142 ms Memory used: 384 KB
n = 10000, m = 10000 длина слов от 1 до 6	Classical LCS: Length: 547 Time taken: 3261 ms Memory used: 390588 KB	Hirschberg LCS: Length: 547 Time taken: 4648 ms Memory used: 736 KB

Выводы

В ходе выполнения данного курсового проекта я познакомился с эффективной по памяти реализацией решения задачи поиска наибольшей общей последовательности — с алгоритмом Хиршберга.

Было проведено сравнение классического решения задачи НОП и алгоритма Хиршберга. На практике видно, что алгоритм Хиршберга потребляет существенно меньше памяти при больших объёмах данных.