Лабораторная работа № 3 по курсу дискртного анализа: Поиск образца в строке

Выполнил студент группы М8О-208Б-20 *Ядров Артем*.

Условие

Кратко описывается задача:

1. Необходимо реализовать один из стандартных алгоритмов поиска образцов для указанного алфавита.
2. *Вариант алгоритма:* Поиск одного образца при помощи алгоритма Кнута-Морриса-Пратта.
3. *Вариант алфавита:* Слова не более 16 знаков латинского алфавита (регистронезависимые).
4. Запрещается реализовывать алгоритмы на алфавитах меньшей размерности, чем указано в задании.

Метод решения

Алгоритм Кнута-Морисса-Прутта
Требуется реализовать алгоритм Кнута-Морриса-Пратта для поиска подстроки в строке. Учитывая, что алфавит состоит из регистронезависимых слов не более 16 знаков, нужно уметь правильно представлять переводы строки, пробелы и табуляции.
Алгоритм Кнута-Морриса-Пратта прикладывает образец к тексту и начинает делать сравнение с левого конца. В случае полного совпадения, было найдено вхождение, сдвигаем образец на один символ вправо. Если же есть несовпадения, то мы делаем сдвиг по особому правилу, в отличии от алгоритма наивного поиска, который всегда сдвигает на один символ.
Для каждой позиции i определим $sp_i(P)$ как длину наибольшего собственного суффикса $P[1..i]$, который совпадает с префиксом P, причём символы в позициях $i+1$ и $sp_i(P)+1$ различны.
Для каждой позиции i определим $Z_i(P)$ как длину префикса строки $P[i..|P|]$, который совпадает с префиксом P. Причем $Z_0(P)$ принято считать равным нулю. Набор таких значений называется Z-функцией строки P. Z-функция является известным алгоритмом и может быть вычеслена за линейное время от длины строки.
Значения $Z_j(P)$ соответствуют такому $sp_i(P)$, что $i = j + Z_j(P) - 1$. Таким образом вычисление всех $sp_i(P)$ имеет сложность $O(n)$, где n - длина образца.
Будем делать сдвиг, используя вычисленное в каждой позиции значение $sp_i(P)$. Если при сравнении было найдено несовпадение в позиции $i+1$, то мы можем сделать сдвиг на $i - sp_i(P)$, не теряя вхождений.

Алгоритм Кнута-Морисса-Пратта сравнивает каждый символ не более двух раз, то есть совершает не более $2 * m$ сравнений символов, где m - длина текста. Учитывая вычисление Z-функции, сложность алгоритма составит $O(n + m)$.

Описание программы

Разобьем алгоритм на следующие шаги:

1. Реализация вспомогательных структур и функций
2. Реализация вычисления Z и SP-функции
3. Реализация алгоритма Кнута-Морисса-Пратта с использованием SP-функции
4. Реализация правильного ввода и поиска подстроки в строке

Создадим вспомогательную структуру $TWord$, которая будет хранить в себе английское слово - символ нашего алфавита, также его номер в строке и номер строки в тексте. Также будем хранить размер слова и его хэш для более быстрой реализации сравнения слов.

```
#ifndef INC_4LAB_SRC_TWORD_HPP
#define INC_4LAB_SRC_TWORD_HPP

#include <iostream>
#include <vector>

const unsigned short MAX_WORD_SIZE = 17;
const unsigned short ALPHABET_SIZE = 26;

struct TWord {
    TWord();

    friend bool operator==(const TWord &lhs, const TWord &rhs);

    friend bool operator!=(const TWord &lhs, const TWord &rhs);

    char Word[MAX_WORD_SIZE];
    unsigned int StringID;
    unsigned int WordID;
    unsigned int Size;
    unsigned int Hash;
};

std::vector<int> ZFunction(const std::vector<TWord> &string);
```

```
std::vector<int> SPFunction(const std::vector<TWord> &string);

std::vector<int>
Search(const std::vector<TWord> &pattern, const std::vector<TWord> &text,
↪  const std::vector<int> &sp);

std::vector<int> NaiveSearch(const std::vector<TWord> &pattern, const
↪  std::vector<TWord> &text);

#endif //INC_4LAB_SRC_TWORD_HPP
```

Реализацию функций, описанных выше, я поместил в отдельный файл.

```
#include "TWord.hpp"

TWord::TWord() : WordID(0), StringID(0), Hash(0), Size(0) {}

bool operator==(const TWord &lhs, const TWord &rhs) {
    if (lhs.Hash != rhs.Hash) {
        return false;
    }
    for (int i = 0; i < lhs.Size; ++i) {
        if (lhs.Word[i] != rhs.Word[i]) {
            return false;
        }
    }
    return true;
}

bool operator!=(const TWord &lhs, const TWord &rhs) {
    return !(lhs == rhs);
}

std::vector<int> ZFunction(const std::vector<TWord> &string) {
    int n = (int) string.size();
    std::vector<int> z(n);
    for (int i = 1, left = 0, right = 0; i < n; ++i) {
        if (i <= right) {
            z[i] = std::min(right - i + 1, z[i - left]);
        }
        while (i + z[i] < n and string[z[i]] == string[i + z[i]]) {
            ++z[i];
```

```
        }
        if (i + z[i] - 1 > right) {
            left = i, right = i + z[i] - 1;
        }
    }
    return z;
}

std::vector<int> SPFunction(const std::vector<TWord> &string) {
    std::vector<int> z = ZFunction(string);
    int n = (int) z.size();
    std::vector<int> sp(n);
    for (int i = n - 1; i > 0; --i) {
        sp[i + z[i] - 1] = z[i];
    }
    return sp;
}

std::vector<int>
Search(const std::vector<TWord> &pattern, const std::vector<TWord> &text,
↪   const std::vector<int> &sp) {
    size_t patternSize = pattern.size();
    size_t textSize = text.size();
    std::vector<int> ans;
    if (patternSize > textSize) {
        return ans;
    }
    unsigned int i = 0;
    while (i < textSize - patternSize + 1) {
        unsigned int j = 0;
        while (j < patternSize and text[i + j] == pattern[j]) {
            ++j;
        }
        if (j == patternSize) {
            ans.emplace_back(i);
        } else if (j > 0 and j > sp[j - 1] + 1) {
            i = i + j - sp[j - 1] - 1;
        }
        ++i;
    }
    return ans;
}
```

```
std::vector<int> NaiveSearch(const std::vector<TWord> &pattern, const
↪  std::vector<TWord> &text) {
    size_t patternSize = pattern.size();
    size_t textSize = text.size();
    std::vector<int> ans;
    if (patternSize > textSize) {
        return ans;
    }
    unsigned int i = 0;
    while (i < textSize - patternSize + 1) {
        unsigned int j = 0;
        while (j < patternSize and text[i + j] == pattern[j]) {
            ++j;
        }
        if (j == patternSize) {
            ans.emplace_back(i);
        }
        ++i;
    }
    return ans;
}
```

И, наконец, в финальном файле содержится ввод и вызов остальных функций. Аккуратно, при просмотре у опытного программиста может случиться инфаркт.

```
#include "TWord.hpp"

inline void Clear(TWord &word) {
    word.Hash = 0;
    word.Size = 0;
}

int main() {
    std::ios_base::sync_with_stdio(false);
    std::cin.tie(nullptr);
    std::cout.tie(nullptr);
    std::vector<TWord> pattern;
    std::vector<TWord> text;
    int start = 0;
    unsigned short ind = 0;
    TWord current;
    std::vector<int> sp;
```

```
    std::string buffer;
    getline(std::cin, buffer);
    for (auto &c: buffer) {
        if (c == ' ' or c == '\t') {
            if (ind > 0) {
                pattern.emplace_back(current);
            }
            Clear(current);
            ind = 0;
        } else {
            current.Word[ind] = toupper(c);
            current.Hash = current.Hash * ALPHABET_SIZE + current.Word[ind]
↪   - 'A';
            ++ind;
        }
    }
    if (ind > 0) {
        pattern.emplace_back(current);
        Clear(current);
        ind = 0;
    }
    sp = SPFunction(pattern);
    text.reserve(2 * pattern.size());
    current.WordID = 1;
    current.StringID = 1;
    while (getline(std::cin, buffer)) {
        for (auto &c: buffer) {
            if (c == '\t' or c == ' ') {
                if (ind > 0) {
                    text.emplace_back(current);
                    if (text.size() > 2 * pattern.size()) {
                        Search(pattern, text, sp, start);
                        text.erase(text.begin(), text.begin() + (int)
↪   pattern.size());
                        text.reserve(2 * text.size());
                    }
                    ind = 0;
                    ++current.WordID;
                    Clear(current);
                }
            } else {
                current.Word[ind] = toupper(c);
```

```
                current.Hash = current.Hash * ALPHABET_SIZE +
↪  current.Word[ind] - 'A';
                ++ind;
            }
        }
        if (ind > 0) {
            text.emplace_back(current);
            if (text.size() > 2 * pattern.size()) {
                Search(pattern, text, sp, start);
                text.erase(text.begin(), text.begin() + (int)
↪  pattern.size());
                text.reserve(2 * text.size());
            }
        }
        current.WordID = 1;
        ++current.StringID;
        Clear(current);
        ind = 0;
    }
    if (ind > 0) {
        text.emplace_back(current);
    }
    Search(pattern, text, sp, start);
    return 0;
}
```

Так как мы не можем хранить ВЕСЬ текст, то я решил воспользоваться буффером: хранить текст, превышающий 2 длины шаблона. После того, как буффер заполнится, воспользуемся алгоритмом Кнута-Морисса-Пратта, затем удалим первую половину буффера, при этом запоминая, на каком элементе мы остановились. Также во время чтения вычисляется полиномиальная хэш-функция, которая помогает быстро сравнивать строки.

Дневник отладки

Во время реализации я столкнулся с большими проблемами:

1. Проблема с памятью. Изначально я не использовал буффер, а вызывал алгоритм поиска сразу для всего текста, что крайне негативно сказывалось на затрачиваемой памяти.

2. Проблема со временем.

   (a) Изначально *TWord* был классом с закрытыми полями. Поэтому пришлось реализовывать неэффективные по времени методы для работы с данным

классом. Чтобы ускорить программу, я решил отказаться от безопасности и сделал класс структурой с открытыми полями.

(b) Также сравнение двух символов алфавита (двух регистронезависисых слов) стоило очень много времени. Изначально я решил сохранять текст в своем первоначальном виде (на случай, если придется дополнительно его выводить куда-либо), поэтому во время сравнения я приводил все символы к верхнему регистру. Это операция, хоть на первый взгляд и не является существенной, вызывается довольно часто. Поэтому я решил сразу приводить к верхнему регистру.

(c) Опять сравнение. Я сравнивал все 16 символов слова, что, естественно, негативно сказывалось на времени. Поэтому я решил добавить поле *Size* для струкутры *TWord*. Это несколько ускорило операцию сравнения.

(d) Финальное ускорение сравнения. Я решил хранить хэш слова. Конечно, размер тоже является своего рода хэшэм, но он имеет достаточно большое количество колизий. Я решил воспользоваться тривиальным, но эффективным биномиальным хэшэм.

(e) Изначально ввод происходил посимвольно. Посимвольный ввод это достаточно дорогая, как я выяснил позднее, операция. Легче считать построчно, а затем обрабатывать строку, нежели посимвольно считывать текст.

Тест производительности

Реализованный алгоритм Кнута-Морриса-Пратта с испольнованием прфикс функции сравнивается с наивным алгоритмом поиска. Замеры производятся на тестах из 10^3 слов, 10^4 или 10^5. Длина образца в первом тесте - 10, во втором - 25, в третьем - 100

```
[yadroff@fedora src]$ ./benchmark < tests/03.t
==============START============
Naive: 0.822 ms
KMP: 0.049 ms
==============END==============
[yadroff@fedora src]$ ./benchmark < tests/04.t
==============START============
Naive: 5.418 ms
KMP: 0.564 ms
==============END==============
[yadroff@fedora src]$ ./benchmark < tests/05.t
==============START============
Naive: 25.715 ms
KMP: 2.412 ms
==============END==============
```

Видно, что наивный алгоритм поиска почти на порядок проигрывает алгоритму Кнута-Морриса-Пратта. Классический алгоритм допускает лишние сравнения на этапе поиска образца в тексте, а алгоритм с применение префикс функции - нет. Обработка таких сравнений длиться дольше, чем предпроцессинг префикс функции.

Выводы

Во время выполнения лабораторной работы я изучил алгоритм Кнута-Морриса-Пратта с предпроцессингом через префикс функцию.
Задачи поиска подстроки часто встречаются в жизни, один из очевидных примеров это поиск контента в Интернете по ключевому слову или по ключевой фразе.
Наивный алгоритм поиска нельзя использовать для поиска, т.к. он действует слишком медленно. С другой стороны, КМП не может эффективно находить несколько образцов в тексте, но с этим хорошо справляется алгоритм Ахо-Корасика.