**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**Кафедра МО ЭВМ**

**ОТЧЕТ**

# по лабораторной работе №2

**по дисциплине «Построение и анализ алгоритмов» Тема: Коммивояжёр**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3341 |  | Ягудин Д.Р. |
| Преподаватель |  | Фирсов М.А. |

Санкт-Петербург 2021

# Цель работы.

Применить на практике знания о методах решения задачи коммивояжёра (TSP) с использованием точных и приближённых алгоритмов. Реализовать точный метод на основе динамического программирования (итеративная реализация) для нахождения оптимального маршрута. Реализовать приближённый алгоритм "Ближайший сосед" (АЛШ-2) для нахождения приближённого решения задачи коммивояжёра. Сравнить результаты работы точного и приближённого методов на различных тестовых данных.

# Задание.

Напишите программу, решающую задачу коммивояжера. Нужно найти кратчайший маршрут, который проходит через все заданные города ровно один раз и возвращается в исходный город. Не все города могут быть напрямую связаны друг с другом.

**Входные данные:**

* n - количество городов (5 ≤ n ≤ 15).
* Матрица расстояний между городами размером n×n , где graph[i][j] обозначает расстояние от города i до города j. Если graph[i][j]=0  (и i ≠ j ), это означает, что прямого пути между городами нет.

**Выходные данные:**

* Минимальная стоимость маршрута, проходящего через все города и возвращающегося в начальный город.
* Оптимальный путь в виде последовательности посещаемых городов, начинающегося и заканчивающегося в начальном городе.
* Если такого пути не существует, вывести "no path".

Sample Input 1:

5

0 1 13 23 7

12 0 15 18 28

21 29 0 33 28

23 19 34 0 38

5 40 7 39 0

Sample Output 1:

78

0 4 2 3 1 0

Sample Input 2:

3

0 1 0

1 0 1

0 1 0

Sample Output 2:

no path

# Ход работы:

1. Произведён анализ задания.
2. Был реализован алгоритм вычисления префикс-функции:
   1. Сперва создается массив, в который в дальнейшем будут записаны значения префикс-функции. Размер массива равен длине заданной строки S, для которой считается префикс-функция. Элемент π[0] равен 0.
   2. Для подсчёта текущего значения π[i] заводится переменная j, обозначающая длину текущего рассматриваемого образца. Изначально j = π[i-1].
   3. Сравниваются символы S[j] и S[i]. Если они совпадают — то π[i] = j+1 и алгоритм переходит к следующему индексу i+1. Если же символы отличаются, то j становится равна π[j-1], и данный шаг повторяется с начала.
   4. Если алгоритм дошел до j=0 и так и не были найдены совпадения, то π[i] = 0 и осуществляется переход к следующему индексу i+1.
3. Был реализован алгоритм Кнута-Морриса-Пратта:
   1. Сперва считается префикс-функция для строки вида P + #, где P – шаблон строки, которую необходимо найти в тексте T.
   2. После этого с помощью цикла for программа проходит по тексту, сравнивая символы. При этом сдвиг происходит не на один символ, как при наивном сравнении, а на значение префикс-функции для данного символа.
   3. Постепенно проходясь по тексту программа записывает в структуру vector из стандартной библиотеки языка C++ индексы, в которых текущая рассматриваемая подстрока текста совпадает с шаблонной строкой.
4. Сложность алгоритма вычисления префикс-функции как по памяти, так и по времени линейна — O(|S|), где |S| – длина строки, для которой нужно вычислить префикс-функцию.
5. Сложность алгоритма Кнута-Морриса-Пратта

по памяти — O(|P|), где |P| – длина строки шаблона, для которой вычисляется префикс-функция. По времени же сложность можно оценить как O(|P| + |T|), где |P| – длина строки шаблона, а |T| - длина текста, в котором ведется поиск. Фактически сложность состоит из построения префикс-функции для шаблона и дальнейшего прохода по всем символам текста.

1. Задача 1 была решена простым применением алгоритма Кнута-Морриса- Пратта.
2. Задача 2 была решена с помощью алгоритма Кнута-Морриса-Пратта с удвоенной строкой текста T, так как в такой строке точно встретится шаблон P, если текст действительно является циклическим сдвигом шаблона.
3. Для удобства выбора заданий, были разработаны ключи командной строки для запуска соответствующих заданий: ключ «*-kmp*» запускает выполнение задачи 1, ключ «*-cycle*» запускает выполнение задачи 2.
4. Были написаны тесты с использованием библиотеки Catch2 для функций программы:
   1. Была протестирована функция считывания строк.
   2. Была протестирована корректная работа функции вычисления префикс-функции для заданной строки — как для непустой, так и для пустой.
   3. Была протестирована функция, реализующая алгоритм Кнута- Морриса-Пратта. Функция была протестирована на разных входных данных, в том числе и тех, когда шаблон или текст являются пустыми строками.
   4. Была протестирована функция поиска циклического сдвига. Функция была протестирована на разных входных данных, в том числе и тех, когда шаблон или текст являются пустыми строками.
   5. Тесты, описанные в данном разделе, представлены в разделе Тестирование функций.
5. Код разработанной программы расположен в Приложении А.

# Описание функций и структур данных.

1. std::vector<int> ComputePrefixFunction(const std::string& pattren) - вычисляет значение префикс-функции для заданной строки. Возвращает массив значений префикс-функции, pattern – строка для которой мы высчитываем префикс-функцию.
2. std::vector<int> KMP search(const std::string& text, const std::string& pattern) - реализует алгоритма Кнута-Морриса-Пратта. Возвращает массив с индексами всех вхождений шаблона в текст, text – строка в которой ищется подстрока, pattern – подстрока которую нужно найти в строке text.
3. int *main(int argc, char\* argv)* - функция, в которой происходит считывание входных данных и запуск выполнения задания выбранного флагом при запуске программы, argc – количество аргументов, argv – массив аргументов .

# Тестирование.

Таблица 1 – Пример работы программы в задании №1

|  |  |  |
| --- | --- | --- |
| № п/п | Входные данные | Выходные данные |
| 1. | ab abab | 0,2 |
| 2. | ab  zxcx | -1 |
| 3. | ab abababababab | 0,2,4,6,8,10 |

Таблица 2 – Пример работы программы в задании №2

|  |  |  |
| --- | --- | --- |
| № п/п | Входные данные | Выходные данные |
| 1. | defabc abcdef | 3 |
| 2. | qew  cfv | -1 |
| 3. | aaa aaa | 0 |

# Выводы.

Были применены на практике знания о построение алгоритма Кнута- Морриса-Пратта. Была реализован алгоритм вычисления префикс-функции на языке программирования C++. Был реализован алгоритм Кнута-Морриса- Пратта на языке программирования C++. С помощью реализованных алгоритмов были решены два задания: поиск всех подстрок по заданному шаблону в заданном тексте и проверка, является ли одна строка циклическим сдвигом другой.

**ПРИЛОЖЕНИЕ А**

Название файла: main.cpp

#include <iostream>

#include <vector>

#include <string>

// Функция для вычисления префикс-функции (pi-функции)

std::vector<int> computePrefixFunction(const std::string& pattern) {

    std::cout << "-----------------------------\n";

    std::cout << "Calculating prefix function for pattern: " << pattern << '\n';

    int m = pattern.size();

    std::vector<int> pi(m, 0);

    int j = 0;

    for (int i = 1; i < m; i++) {

        while (j > 0 && pattern[i] != pattern[j]) {

            j = pi[j - 1];

        }

        if (pattern[i] == pattern[j]) {

            j++;

        }

        pi[i] = j;

        std::cout << "pi[" << i << "] = " << pi[i] << '\n';

    }

    std::cout << "Ready prefix: ";

    for (int x : pi) std::cout << x << " ";

        std::cout <<"\n-----------------------------\n";

    return pi;

}

// Функция КМП

std::vector<int> KMPsearch(const std::string& text, const std::string& pattern) {

    int n = text.size(), m = pattern.size();

    std::vector<int> pi = computePrefixFunction(pattern);

    std::vector<int> result;  // Индексы вхождений шаблона в текст

    int j = 0; // Указатель на pattern

    std::cout << "-----------------------------\n";

    std::cout << "Start finding " << pattern << " in " << text << '\n';

    for (int i = 0; i < n; i++) {

        while (j > 0 && text[i] != pattern[j]) {

            std::cout << "Mismatch: " << text[i] << " != " << pattern[j]  << ", rollback j = " << pi[j - 1] << '\n';

            j = pi[j - 1]; // Откат

        }

        if (text[i] == pattern[j]) {

            j++; // Совпадение продолжается

            std::cout << "Match: " << text[i] << " (i=" << i << ", j=" << j << ")" << '\n';

        }

        if (j == m) { // Полное совпадение

            std::cout << "Find on index " << i - m + 1 << '\n';

            result.push\_back(i - m + 1);

            j = pi[j - 1]; // Продолжаем поиск

        }

    }

    std::cout << "-----------------------------\n";

    return result;

}

int main(int argc, char\* argv[]) {

    if (argc == 0) {

        std::cout << "No arguments provided." << '\n';

        return 0;

    }

    std::string A, B;

    std::cin >> A >> B;

    std::vector<int> matches;

    std::string arg = argv[1];

    if (arg== "--first") {

        matches = KMPsearch(B, A);

        if (matches.size() == 0){

            std::cout << -1 << '\n';

        }else{

            std::string out;

            for (int pos : matches) {

                out += std::to\_string(pos);

                out += ",";

            }

            out.pop\_back();

            std::cout << out <<'\n';

        }

        return 0;

    }

    if (arg == "--second") {

        if (A.size() != B.size()) {

            std::cout << -1 << '\n';

            return 0;

        }

        std::string AA = A + A; // Дублируем строку A

        matches = KMPsearch(AA, B); // Ищем B в A+A

        if (matches.size() != 0) {

            std::cout << matches[0] << '\n';

        }else{

            std::cout << -1 << '\n';

        }

        return 0;

    }

    return 0;

}