Московский авиационный институт (национальный исследовательский университет)

Институт №8 «Компьютерные науки и прикладная математика»

Кафедра 806 «Вычислительная математика и программирование»

Лабораторная работа по курсу "Дискретный анализ" №5

Студент: Ахметшин Б. Р.

Преподаватель: Макаров Н. К.

Группа: М8О-303Б-22

Дата: _____

Оценка: _____

Подпись: _____

Москва 2024 г.

Оглавление

Постановка задачи

Реализовать поиск подстрок в тексте с использование суффиксного дерева. Суффиксное дерево можно построить за $O(n^2)$ наивным методом.

Формат ввода

Текст располагается на первой строке, затем, до конца файла, следуют строки с образцами.

Формат вывода

Для каждого образца, найденного в тексте, нужно распечатать строчку, начинающуюся с последовательного номера этого образца и двоеточия, за которым, через запятую, нужно перечислить номера позиций, где встречается образец в порядке возрастания.

Алгоритм решения

Применим алгоритм Укконена для построения суффиксного дерева.

Поиск осуществим следующим образом:

1. Добавим к строке сентинелл.
2. Начинаем с корня.
3. Берем очередной символ паттерна и смотрим, какое ребро начинается с этого символа. Если следующего символа нет — переходим к пункту 5.
4. Идем по этому ребру, сравнивая символы на нем с паттерном.
    i. Если символ на ребре не совпал с паттерном — завершаем поиск и возвращаем пустое множество.
    ii. Если ребро или паттерн закончились, переходим к пункту 3.
5. Теперь ясно, что паттерн встречается в тексте, а все листья, выходящие из текущей вершины, в суффиксных индексах содержат индексы вхождений. Проходим *bfs* по

поддереву, если текущая вершина — лист, сохраняем ее суффиксный индекс в множество.

Исходный код

```
#ifndef TYPE_ALIASES_HPP
#define TYPE_ALIASES_HPP

#include <cinttypes>
#include <limits>
#include <memory>
#include <vector>
#include <string>

namespace lab {
    // Integer type aliases
    using u64 = uint64_t;
    using i64 = int64_t;
    using u32 = uint32_t;
    using i32 = int32_t;
    using u16 = uint16_t;
    using i16 = int16_t;
    using u8 = uint8_t;
    using i8 = int8_t;

    template <class T>
    using limit = std::numeric_limits<T>;

    // Alias for shared_ptr of basic types
    using CharPtr = std::shared_ptr<char>;
    using U64Ptr = std::shared_ptr<u64>;
    using I64Ptr = std::shared_ptr<i64>;
}

#endif // TYPE_ALIASES_HPP


#ifndef SUFFIX_NODE_HPP
#define SUFFIX_NODE_HPP

#include <map>
#include <memory>

namespace lab {

    // Forward declaration of SuffixTree
    class SuffixTree;

    // SuffixNode class representing a node in the suffix tree
```

```
    class SuffixNode {
    public:
        using SuffixNodePtr = std::shared_ptr<SuffixNode>;
        using ChildrenMap = std::map<char, SuffixNodePtr>;

        // Constructors
        SuffixNode();
        SuffixNode(u64 start, U64Ptr end);

        // Node properties
        u64 getStart() const;
        u64 getEnd() const;
        ChildrenMap& getChildren();
        SuffixNodePtr getSuffixLink();
        void setSuffixLink(SuffixNodePtr node);

        // Suffix Index
        void setSuffixIndex(u64 index);
        u64 getSuffixIndex() const;

    private:
        u64 start;
        U64Ptr end;
        ChildrenMap children;
        SuffixNodePtr suffixLink; // Link to another node in the tree
        u64 suffixIndex;          // Suffix index for leaf nodes (default: -1 if
not a leaf)

        friend SuffixTree;
    };
}

#endif // SUFFIX_NODE_HPP

#ifndef SUFFIX_TREE_HPP
#define SUFFIX_TREE_HPP

#include <string>
#include <vector>
#include <set>

namespace lab {

    class SuffixTree {
    public:
        using StringPtr = std::shared_ptr<std::string>;
        using SuffixNodePtr = SuffixNode::SuffixNodePtr;
```

```
        // Constructors
        SuffixTree(const std::string& text);

        // Public interface
        void buildTree(const std::string& text);
        std::set<u64> searchPattern(const std::string& pattern);
        static std::pair<u64, std::vector<u64>> findLCS(const std::string& S1,
const std::string& S2);

    private:
        // Internal helper functions
        void extendTree(u64 pos);
        void setSuffixIndexByDFS(SuffixNodePtr node, u64 labelHeight);
        static void findLCSUtil(
            SuffixNodePtr node,
            u64 depth,
            u64& maxLength,
            u64 splitPoint,
            std::map<u64, u64>& nodeCSLengths
        );

        // Tree properties
        StringPtr text;                         // The input string
        SuffixNodePtr root;        // Root of the suffix tree
        SuffixNodePtr activeNode;  // Active node for construction
        u64 activeEdge;
        u64 activeLength;
        u64 remainingSuffixCount;
        U64Ptr leafEnd;
        U64Ptr rootEnd;
        U64Ptr splitEnd;
        u64 size; // Size of the input string

        friend std::ostream& operator<<(std::ostream& os, SuffixTree const& t);

    };

    std::ostream& operator<<(std::ostream& os, SuffixTree const& t);
}

#endif // SUFFIX_TREE_HPP

namespace lab {

    SuffixNode::SuffixNode()
        :   start(0ul),
            end(nullptr),
            suffixLink(nullptr),
```

```
            suffixIndex(limit<u64>::max()) {}

    SuffixNode::SuffixNode(u64 start, U64Ptr end)
        :   start(start),
            end(end),
            suffixLink(nullptr),
            suffixIndex(limit<u64>::max()) {}

    u64 SuffixNode::getStart() const {
        return start;
    }

    u64 SuffixNode::getEnd() const {
        return end ? *end : 0;
    }

    SuffixNode::ChildrenMap& SuffixNode::getChildren() {
        return children;
    }

    SuffixNode::SuffixNodePtr SuffixNode::getSuffixLink() {
        return suffixLink;
    }

    void SuffixNode::setSuffixLink(SuffixNodePtr node) {
        suffixLink = node;
    }

    void SuffixNode::setSuffixIndex(u64 index) {
        suffixIndex = index;
    }

    u64 SuffixNode::getSuffixIndex() const {
        return suffixIndex;
    }
}

#include <queue>
#include <functional>
#include <iostream>

namespace lab {

    SuffixTree::SuffixTree(const std::string& text)
        :   text(std::make_shared<std::string>(text)),
            size(text.size()) {
        buildTree(text);
    }
```

```
    void SuffixTree::buildTree(const std::string& text) {
        root = std::make_shared<SuffixNode>(
            limit<u64>::max(),
            std::make_shared<u64>(limit<u64>::max())
        );
        activeNode = root;
        activeEdge = limit<u64>::max();
        activeLength = 0;
        remainingSuffixCount = 0;
        leafEnd = std::make_shared<u64>(limit<u64>::max());
        rootEnd = nullptr;
        splitEnd = nullptr;

        for (u64 i = 0; i < size; ++i) {
            extendTree(i);
        }

        setSuffixIndexByDFS(root, 0);
    }

    void SuffixTree::extendTree(u64 pos) {
        // Set the end for leaf nodes
        *leafEnd = pos;

        // Number of suffixes to be added
        remainingSuffixCount++;

        SuffixNodePtr lastNewNode = nullptr;

        // Iterate while there are still suffixes to add
        while (remainingSuffixCount > 0) {
            // If active length is 0, the current character is the active edge
            if (activeLength == 0) {
                activeEdge = pos;
            }

            char currentChar = (*text)[pos];
            char activeChar = (*text)[activeEdge];

            // Check if the current character exists in the active node's
children
            if (activeNode->getChildren().find(activeChar) == activeNode-
>getChildren().end()) {
                // No such edge exists, create a new leaf node
                activeNode->getChildren()[activeChar] =
std::make_shared<SuffixNode>(pos, leafEnd);
```

```
                // Link the last created internal node to this one if necessary
                if (lastNewNode != nullptr) {
                    lastNewNode->setSuffixLink(activeNode);
                    lastNewNode = activeNode->getChildren()[activeChar];
                }
            } else {
                // There is an edge, find the next node
                SuffixNodePtr nextNode = activeNode->getChildren()[activeChar];

                // Check if we are in the middle of an edge
                u64 edgeLength = nextNode->getEnd() - nextNode->getStart() + 1;
                if (activeLength >= edgeLength) {
                    // Move to the next node
                    activeEdge += edgeLength;
                    activeLength -= edgeLength;
                    activeNode = nextNode;
                    continue;
                }

                // The character is already in the edge, rule 3 (extension ends)
                if ((*text)[nextNode->getStart() + activeLength] == currentChar)
{
                    // We increment the active length and break
                    activeLength++;
                    if (lastNewNode != nullptr) {
                        lastNewNode->setSuffixLink(activeNode);
                        lastNewNode = nullptr;
                    }
                    break;
                }

                // Split the edge, create a new internal node
                splitEnd = std::make_shared<u64>(nextNode->getStart() +
activeLength - 1);
                SuffixNodePtr splitNode = std::make_shared<SuffixNode>(nextNode-
>getStart(), splitEnd);
                activeNode->getChildren()[activeChar] = splitNode;

                // Create a new leaf node
                splitNode->getChildren()[currentChar] =
std::make_shared<SuffixNode>(pos, leafEnd);

                // Adjust the next node's start position
                nextNode->start += activeLength;
                splitNode->getChildren()[(*text)[nextNode->start]] = nextNode;

                // Link last internal node to the new split node
                if (lastNewNode != nullptr) {
```

```
                    lastNewNode->setSuffixLink(splitNode);
                }
                lastNewNode = splitNode;
            }

            // Decrement the remaining suffix count
            remainingSuffixCount--;

            // Update the active point
            if (activeNode == root && activeLength > 0) {
                activeLength--;
                activeEdge = pos - remainingSuffixCount + 1;
            } else if (activeNode != root) {
                activeNode = activeNode->getSuffixLink() ? activeNode-
>getSuffixLink() : root;
            }
        }
    }

    // Depth-First Search to assign suffix indices to each leaf node
    void SuffixTree::setSuffixIndexByDFS(SuffixNode::SuffixNodePtr node, u64
labelHeight) {
        if (!node) return;

        // If it's a leaf, assign the suffix index
        if (node->getChildren().empty()) {
            u64 suffixIndex = size - labelHeight;
            node->setSuffixIndex(suffixIndex); // Correctly setting the suffix
index
            return;
        }

        // Traverse all children
        for (auto& [key, child] : node->getChildren()) {
            setSuffixIndexByDFS(child, labelHeight + (child->getEnd() - child-
>getStart() + 1));
        }
    }

    // Searches for a pattern in the suffix tree
    std::set<u64> SuffixTree::searchPattern(const std::string& pattern) {
        if (pattern == "") {
            return {};
        }
        SuffixNodePtr currentNode = root; // Start from the root node
        u64 patternIndex = 0;             // Track the current index of the
pattern
```

```
        // Traverse while there are characters left in the pattern
        while (patternIndex < pattern.size()) {
            char currentChar = pattern[patternIndex];

            // Check if the current character exists in the current node's
children
            if (currentNode->getChildren().find(currentChar) == currentNode-
>getChildren().end()) {
                // The current character is not found among the children,
pattern does not exist
                return {};
            }

            // Move to the next node
            SuffixNodePtr nextNode = currentNode->getChildren()[currentChar];
            u64 edgeStart = nextNode->getStart();
            u64 edgeEnd = nextNode->getEnd();

            // Compare the pattern characters with the edge characters
            for (u64 i = 0; i <= (edgeEnd - edgeStart) && patternIndex <
pattern.size(); ++i) {
                // If characters don't match, pattern is not found
                if ((*text)[edgeStart + i] != pattern[patternIndex]) {
                    return {};
                }
                patternIndex++; // Move to the next character in the pattern
            }

            // Move to the next node in the tree
            currentNode = nextNode;
        }

        // If the entire pattern has been successfully traversed, it exists in
the text
        std::set<u64> indexes;
        std::queue<SuffixNodePtr> order;
        order.push(currentNode);
        while (!order.empty()) {
            auto& node = order.front();
            order.pop();
            if (node->getChildren().empty()) {
                indexes.insert(node->suffixIndex);
            }
            for (auto& [ch, child] : node->getChildren()) {
                order.push(child);
            }
        }
        return indexes;
```

```
    }

}
```

```
#include <iostream>
#include <set>

using namespace lab;

int main() {
    std::string text;
    std::getline(std::cin, text);

    SuffixTree tree(text + "$");

    std::string pattern;
    u64 count = 1;
    while (std::getline(std::cin, pattern)) {
        auto indexes = tree.searchPattern(pattern);
        if (!indexes.empty()) {
            std::cout << count << ": ";
            auto i = indexes.begin();
            std::cout << *(i++) + 1;
            while (i != indexes.end()) {
                std::cout  << ", " << *(i++) + 1;
            }
            std::cout << "\n";
        }
        ++count;
    }

    return 0;
}
```

Тесты

Input	Output
abcdabc abcd bcd bc	1: 1 2: 2 3: 2, 6

Input	Output
aaa a	1: 1, 2, 3

Вывод

В ходе лабораторной работы я реализовал алгоритм Укконена, а также научился искать паттерн в тексте при помощи суффиксного дерева.