Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №5 по курсу «Дискретный анализ»

Студент: М. М. Парфенов
Преподаватель: С. А. Михайлова
Группа: М8О-301Б-22

Дата:
Оценка:
Подпись:

Москва, 2025

Лабораторная работа №5

Задача: Найти самую длинную общую подстроку двух строк с использованием суфф. дерева.

1 Описание

Суффиксное дерево — это структура данных, которая представляет все суффиксы заданной строки в виде дерева. Оно позволяет эффективно выполнять операции поиска подстрок. Суффиксное дерево можно использовать для задачи поиска самой длинной общей подстроки двух строк.

Алгоритм

1. Конкатенируем две строки t_1 и t_2, разделяя их уникальным символом-разделителем, который не встречается в исходных строках. Например:

$$s = t_1 + \# + t_2 + \$$$

где # и $ — уникальные разделители.

2. Строим суффиксное дерево для объединённой строки s. Это можно выполнить за время $O(|s|)$, где $|s| = |t_1| + |t_2| + 2$ — длина объединённой строки.

3. Обходим суффиксное дерево, находя самую длинную вершину (или путь), которая содержит суффиксы обеих строк t_1 и t_2. При обходе дерева проверяем, чтобы подстрока, соответствующая текущей вершине, содержала суффиксы, принадлежащие обоим строкам. Это можно проверить по меткам суффиксов.

Сложность

- Построение суффиксного дерева: $O(|s|)$, где $|s| = |t_1| + |t_2| + 2$. - Обход дерева для поиска самой длинной общей подстроки: $O(|s|)$.

Итоговая сложность алгоритма:

$$O(|t_1| + |t_2|)$$

2 Исходный код

Общие сведения

Реализация использует суффиксное дерево, построенное с помощью алгоритма Укконена. Основные особенности алгоритма Укконена:

1. Построение дерева за $O(n)$, где n — длина строки.
2. Использование линейного количества памяти, за счёт хранения только двух чисел (начала и конца) на каждом ребре вместо полного текста.
3. Суффиксные ссылки позволяют переходить от суффикса xa к a, ускоряя обработку.
4. Эвристики, такие как автоматическое продление всех листов путём увеличения общего конца на единицу.

Описание реализации

Данная реализация использует структуру `SuffixTree`, которая:

- Строит суффиксное дерево для строки с использованием массивов и векторов.
- Реализует метод поиска совпадений подстроки в тексте (`findMatches()`).
- Хранит позиции совпадений и извлекает уникальные подстроки максимальной длины.

Основные этапы

1. Построение суффиксного дерева для строки с использованием:

   - Вектора рёбер (`Edge`), которые содержат начало и конец подстроки в исходном тексте и ссылку на дочернюю вершину.
   - Суффиксных ссылок (`suffixLinks`), что обеспечивает переходы между вершинами.

2. Метод поиска совпадений (`findMatches`):

   - Для каждой позиции строки проверяются совпадения с деревом.

- При достижении неудачи выполняется переход по суффиксным ссылкам для эффективного восстановления.

3. Определение самой длинной общей подстроки:

   - Используется массив совпадений для хранения длины совпадения для каждой позиции.
   - Определяется максимальная длина совпадения.
   - Извлекаются уникальные подстроки этой длины.

Оценка сложности

1. Построение суффиксного дерева: $O(n)$, где n — длина строки.
2. Поиск совпадений в тексте: $O(m)$, где m — длина текста.
3. Общее время работы алгоритма: $O(n + m)$.

```
#include <bits/stdc++.h>

struct SuffixTree {
    struct Edge {
        int start;
        std::shared_ptr<int> end;
        int destination;

        Edge(int s, std::shared_ptr<int> e, int d) : start(s), end(e), destination(d)
            {}
    };

    std::string text;
    std::vector< std::vector< Edge > > adjacencyList;
    std::vector<int> suffixLinks;
    std::vector<int> depth;

    int findEdge(int node, char c) {
        for (size_t i = 0; i < adjacencyList[node].size(); ++i) {
            if (c == text[adjacencyList[node][i].start]) {
                return i;
            }
        }
        return -1;
    }

```

```
26      SuffixTree(const std::string& input) : text(input), adjacencyList(input.size() * 2)
            , suffixLinks(input.size() * 2), depth(input.size() * 2) {
27          bool isNewNode = false;
28          int idx = 0;
29          std::shared_ptr<int> endPtr(new int);
30          *endPtr = 0;
31          int commonLength = 0;
32          int currentMatchLength = 0;
33          int currentNode = 0;
34          int currentEdge = -1;
35          int newNode = 0;
36
37          for (size_t i = 0; i < input.size(); ++i) {
38              ++*endPtr;
39              int lastCreatedNode = 0;
40              int currentCreatedNode = -1;
41              while (idx < *endPtr) {
42                  if (currentEdge == -1) {
43                      int nextEdge = findEdge(currentNode, text[idx + commonLength +
                            currentMatchLength]);
44                      if (nextEdge == -1) {
45                          isNewNode = true;
46                      } else {
47                          currentMatchLength = 1;
48                      }
49                      currentEdge = nextEdge;
50                  } else {
51                      if (text[adjacencyList[currentNode][currentEdge].start +
                            currentMatchLength] == text[idx + currentMatchLength +
                            commonLength]) {
52                          ++currentMatchLength;
53                      } else {
54                          isNewNode = true;
55                      }
56                  }
57                  if (currentMatchLength > 0 && adjacencyList[currentNode][currentEdge].
                        start + currentMatchLength == *adjacencyList[currentNode][
                        currentEdge].end) {
58                      currentNode = adjacencyList[currentNode][currentEdge].destination;
59                      currentEdge = -1;
60                      commonLength = commonLength + currentMatchLength;
61                      currentMatchLength = 0;
62                  }
63                  if (isNewNode) {
64                      ++newNode;
65                      if (currentMatchLength == 0 && currentEdge == -1) {
66                          adjacencyList[currentNode].push_back(Edge(idx + commonLength,
                                endPtr, newNode));
67                      } else {
```

```
                    currentCreatedNode = newNode;
                    Edge edge = adjacencyList[currentNode][currentEdge];

                    std::shared_ptr<int> newEndPtr(new int);
                    *newEndPtr = adjacencyList[currentNode][currentEdge].start +
                        currentMatchLength;
                    adjacencyList[currentNode][currentEdge].end = newEndPtr;
                    adjacencyList[currentNode][currentEdge].destination = newNode;
                    edge.start = *newEndPtr;

                    adjacencyList[newNode].push_back(edge);
                    ++newNode;
                    adjacencyList[newNode - 1].push_back(Edge(idx +
                        currentMatchLength + commonLength, endPtr, newNode));
                    depth[newNode - 1] = depth[currentNode] + *adjacencyList[
                        currentNode][currentEdge].end - adjacencyList[currentNode][
                        currentEdge].start;

                    if (lastCreatedNode > 0) {
                        suffixLinks[lastCreatedNode] = currentCreatedNode;
                    }
                    lastCreatedNode = currentCreatedNode;
                }
                int suffixLinkNode = suffixLinks[currentNode];
                int nextEdge = -1;
                int nextCommonLength = depth[suffixLinkNode];
                int nextMatchLength = currentMatchLength + commonLength -
                    nextCommonLength - 1;
                while (nextMatchLength > 0) {
                    nextEdge = findEdge(suffixLinkNode, text[idx + nextCommonLength +
                         1]);
                    int edgeLength = *adjacencyList[suffixLinkNode][nextEdge].end;
                    int edgeStart = adjacencyList[suffixLinkNode][nextEdge].start;
                    if (edgeLength - edgeStart <= nextMatchLength) {
                        nextCommonLength = nextCommonLength + edgeLength - edgeStart;
                        nextMatchLength = nextMatchLength - edgeLength + edgeStart;
                        suffixLinkNode = adjacencyList[suffixLinkNode][nextEdge].
                            destination;
                        nextEdge = -1;
                    } else {
                        break;
                    }
                }
                if (nextEdge != -1) {
                    currentMatchLength = nextMatchLength;
                } else {
                    currentMatchLength = 0;
                }
                commonLength = nextCommonLength;
```

```
                currentNode = suffixLinkNode;
                currentEdge = nextEdge;
                ++idx;
                isNewNode = false;
            } else {
                if (i < input.size() - 1) {
                    break;
                }
            }
        }
    }
}

std::vector<size_t> findMatches(const std::string& query) {
    size_t queryLength = query.size();
    std::vector<size_t> result(queryLength);
    int currentNode = 0;
    int currentEdge = findEdge(0, query[0]);
    int commonLength = 0;
    int currentMatchLength = 0;
    std::stack<int> nodeStack;
    nodeStack.push(0);
    for (size_t i = 0; i < queryLength; ++i) {
        int nextNode = currentNode;
        while (suffixLinks[nextNode] == 0 && !nodeStack.empty()) {
            nextNode = nodeStack.top();
            nodeStack.pop();
        }
        currentNode = suffixLinks[nextNode];
        while (!nodeStack.empty()) {
            nodeStack.pop();
        }
        nodeStack.push(currentNode);
        currentMatchLength = std::max(0, commonLength + currentMatchLength - 1 -
            depth[currentNode]);
        commonLength = depth[currentNode];
        currentEdge = -1;
        while (currentMatchLength >= 0) {
            int nextEdge = findEdge(currentNode, query[i + commonLength]);
            if (nextEdge != -1) {
                int edgeLength = *adjacencyList[currentNode][nextEdge].end -
                    adjacencyList[currentNode][nextEdge].start;
                if (currentMatchLength >= edgeLength) {
                    currentMatchLength -= edgeLength;
                    commonLength += edgeLength;
                    currentNode = adjacencyList[currentNode][nextEdge].destination;
                    nextEdge = -1;
                } else {
                    currentEdge = nextEdge;
```

```
157                     break;
158                 }
159             } else {
160                 break;
161             }
162         }
163         while (currentEdge != -1 && i + commonLength + currentMatchLength <
                queryLength && query[i + commonLength + currentMatchLength] == text[
                adjacencyList[currentNode][currentEdge].start + currentMatchLength]) {
164             ++currentMatchLength;
165             if (adjacencyList[currentNode][currentEdge].start + currentMatchLength
                    == *adjacencyList[currentNode][currentEdge].end) {
166                 int nextNode = adjacencyList[currentNode][currentEdge].destination;
167                 int nextEdge = findEdge(nextNode, query[i + commonLength +
                        currentMatchLength]);
168                 int edgeLength = currentMatchLength;
169                 commonLength += edgeLength;
170                 currentMatchLength = 0;
171                 currentNode = nextNode;
172                 nodeStack.push(currentNode);
173                 currentEdge = nextEdge;
174             }
175         }
176         result[i] = commonLength + currentMatchLength;
177     }
178     return result;
179 }
180 };
181
182 int main() {
183     std::string pattern, text;
184     std::cin >> pattern >> text;
185     pattern += "$";
186     SuffixTree tree(pattern);
187     std::vector<size_t> matches(text.size());
188     matches = tree.findMatches(text);
189
190     size_t maxMatch = *std::max_element(matches.begin(), matches.end());
191     std::cout << maxMatch << '\n';
192     std::set<std::string> uniqueMatches;
193
194     for (size_t i = 0; i < matches.size(); ++i) {
195         if (matches[i] == maxMatch) {
196             std::string substring = text.substr(i, maxMatch);
197             uniqueMatches.insert(substring);
198         }
199     }
200
201     for (const auto& match : uniqueMatches) {
```

```
202         std::cout << match << '\n';
203     }
204 
205     return 0;
206 }
```

3 Консоль

```
 [±main]-> cat input.txt
xabay
xabcbay

 [±main]-> g++ main.cpp
 [±main]-> ./a.out < input.txt
 [±main]->
3
bay
xab
```

4 Выводы

Выполнив данную лабораторную работу, я изучил применение суффиксного дерева для нахождения самой длинной общей подстроки двух строк. Освоил построение суффиксного дерева за $O(n)$, используя алгоритм Укконена, и поиск совпадений подстроки в тексте за $O(m)$, где m — длина текста, с применением статистики совпадений.

Основным преимуществом подхода является его высокая эффективность по времени, что делает его подходящим для задач с большими объемами данных. Однако алгоритм Укконена имеет и недостатки: существенные затраты памяти из-за структуры данных, особенно при работе с длинными строками. Для задач, где требуется просто найти подстроку в строке, альтернативные алгоритмы, такие как Кнута-Морриса-Пратта или Бойера-Мура, могут быть более подходящими.

Список литературы

[1] Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн. *Алгоритмы: построение и анализ, 2-е издание.* — Издательский дом «Вильямс», 2007. Перевод с английского: И. В. Красиков, Н. А. Орехова, В. Н. Романов. — 1296 с. (ISBN 5-8459-0857-4 (рус.))

[2] Список использованных источников оформлять нужно по ГОСТ Р 7.05-2008