28 мая 2024 г. 15:37

Целью данного класса является реализация структуры данных суффиксного дерева с использованием алгоритма Укконена для эффективного хранения и обработки строк.

Класс Suffix_tree представляет собой реализацию суффиксного дерева. В нем содержатся методы для построения дерева из заданной строки, поиска подстроки в дереве, вычисления процента совпадения с другим деревом, вычисление количества вхождения подстроки, вывод дерева и прочие вспомогательные методы

Алгоритм Укконена

Алгоритм Укконена — это эффективный метод для построения суффиксного дерева строки. Он работает за линейное время относительно длины строки и имеет линейную сложность относительно размера алфавита. Основная идея алгоритма заключается в том, чтобы построить суффиксное дерево последовательно, добавляя по одному суффиксу строки на каждом шаге.

Инициализация: Создается пустое суффиксное дерево с корнем, представляющим пустую строку.

Построение дерева: Последовательно добавляются все суффиксы строки. При добавлении каждого суффикса происходит обновление дерева в соответствии с правилами алгоритма.

Обновление дерева: Для добавления нового суффикса проверяется, существует ли уже ребро в дереве, начинающееся с первого символа этого суффикса. Если такого ребра нет, создается новый лист суффикса и добавляется к дереву. Если ребро существует, происходит проверка следующих символов суффикса и соответствующих символов на ребре. Если символы совпадают, продолжается движение вниз по ребру. Если символы не совпадают, ребро разбивается, и к дереву добавляется новый узел с суффиксом.

Суффиксные ссылки: Для эффективности поиска и обновления дерева введены суффиксные ссылки, которые позволяют быстро переходить от одного суффикса к другому.

Реализация алгоритма в коде

Suffix_tree : : update_tree(size_t index)

- Этот метод обновляет суффиксное дерево при добавлении нового суффикса, начинающегося с заданного индекса.
- Пока остаются необработанные суффиксы (переменная remain больше нуля), происходит обработка суффиксов.
- Проверяется, начинается ли новый суффикс с уже существующего ребра в дереве.
- Если такого ребра нет, создается новый лист суффикса и добавляется к дереву.
- Если ребро существует, происходит проверка символов суффикса и символов на ребре, и при необходимости ребро разбивается.
- Обновляются переменные и продолжается обработка следующего суффикса.

```cpp
void Suffix_tree::update_tree(size_t index) {
    last_node = nullptr;
    remain++;
    suffix_end++;

    // Будет выполняться до тех пор, пока все суффиксы не будут обработаны
    while (remain > 0) {
        if (current_length == 0) {
            current_edge = index;
        }

        // Ищем дочерний узел, начиная с символа на текущем ребре
        auto finded_child = current_node->childs.find(line[current_edge]);

        // Если такой узел не нашелся
        if (finded_child == current_node->childs.end()) {
            // Создаем новый узел
            Node* added_word = new Node(index, &suffix_end, root, index - remain + 1);

            // Созданный узел добавляем в дочерние узлы текущего узла
            current_node->childs[line[index]] = added_word;

            // Обновляем ссылку предыдущего созданного узла, если она есть
            if (last_node != nullptr) {
                last_node->suffix_link = current_node;
                last_node = nullptr;
            }
        }
        // Если нашли дочерний узел
        else {
            Node* finded_node = finded_child->second;

            // Спускаемся к узлу
            if (current_length >= suffix_length(finded_node)) {
                current_node = finded_node;
                current_length -= suffix_length(finded_node);
                current_edge += suffix_length(finded_node);
                continue;
            }

            // Спускаемся по ребру
            // Если символ совпал с символом на текущей длине
            if (line[index] == line[finded_child->second->left + current_length]) {
                // Обновляем ссылку предыдущего узла, если есть
                if (last_node != nullptr && current_node != root) {
                    last_node->suffix_link = current_node;
                }
                current_length++;
                break;
            }

            // Если символ не совпал, то разделяем ребро
            Node* new_node = new Node(finded_node->left, new int(finded_node->left + current_length - 1), root, index - remain + 1);

            if (last_node != nullptr) {
                last_node->suffix_link = new_node;
            }

            // Обновляем текущий узел и его дочерние узлы
            current_node->childs[line[current_edge]] = new_node;
            finded_node->left += current_length;

            new_node->childs[line[index]] = new Node(index, &suffix_end, root, index - remain + 1);
            new_node->childs[line[finded_node->left]] = finded_node;
            last_node = new_node;
        }
        // Обновляем переменные после обработки суффикса
        remain--;

        if (current_length > 0 && current_node == root) {
            current_length--;
            current_edge++;
        }
        else if (current_node != root) {
            current_node = current_node->suffix_link;
        }
    }
}
```