

Реализация алгоритмов Ху—Таккера и Гарсия—Уочса для построения оптимальных алфавитных деревьев

Теоретическая часть

Алфавитное дерево — это такое бинарное дерево, при просмотре листьев которого слева направо, соответствующие буквы должны появляться в алфавитном порядке.

1. Алгоритм Ху—Таккера

Пусть имеется порядок листьев V_1, V_2, \dots, V_n и их веса w_1, w_2, \dots, w_n . Эти последовательности назовём соответственно последовательностью узлов и весов.

Если мы комбинируем V_i и V_j , то их отец обозначается через V_{ij} , его вес — через w_{ij} . Если комбинируется V_{ij} и V_k , их отец обозначается через $V_{ij,k}$. В этих

Алгоритм Ху—Таккера сначала строит дерево, не являющееся алфавитным, а затем преобразует его в оптимальное алфавитное дерево.

Понятия:

Два узла в последовательности узлов называются *совместимой парой*, если они соседние или все узлы между ними круглые (т. е. имеют потомков).

Когда комбинируется пара узлов с весами w_i и w_j , вес их отца $w_{ij} = w_i + w_j$ называется *весом этой пары*. Пара с минимальным весом называется *минимальной парой*.

Алгоритм Ху—Таккера строит алфавитное дерево, минимизирующее $\sum w_i l_i$, выполняя следующие шаги.

Комбинирование. По данной последовательности из n узлов с весами w_1, w_2, \dots, w_n строим последовательность из $n - 1$ узла, комбинируя локально минимальную совместимую пару (пара, вес которой меньше, чем вес соседних с ней совместимых пар), заменяя левого сына его отцом и удаляя правого сына из последовательности. Процедура слияния весов продолжается до тех пор, пока не останется один вес.

Определение уровней. Находим номер уровня l_i каждого листа V_i относительно корня (узел с наибольшим значением l_i располагается внизу дерева).

Перестройка. После того, как номера уровней l_1, l_2, \dots, l_n всех листьев определены, применим ним стековый алгоритм. Он заключается в следующих шагах.

Шаг 0. Стек пуст, l_1, l_2, \dots, l_n находятся в очереди.

Шаг 1. Если в стеке меньше двух элементов, перейти к шагу 2. В противном случае проверить, равны ли значения двух верхних элементов стека. Если они различны, перейти к шагу 2, а если равны — к шагу 3.

Шаг 2. Удалить из очереди первый элемент и поместить его на вершину стека. Перейти к шагу 1.

Шаг 3. Пусть l_j – верхний элемент стека, а l_{j-1} — следующий элемент. Заменить l_j и l_{j-1} на $l_j - 1$. Если $l_j - 1 = 0$, остановиться, иначе перейти на шаг 1. (Это означает, что комбинируются узлы V_{j-1} и V_j , а их отец $V_{j-1,j}$ становится узлом уровня $l_j - 1$.)

2. Алгоритм Гарсиа—Уочса

В фазе комбинирования алгоритма Ху—Таккера мы последовательно комбинируем л. м. с. п., при этом рассматриваемые пары могут быть разделены несколькими узлами-отцами. Алгоритм Гарсиа—Уочса устраняет различия между листьями и узлами и располагает узлы в последовательность так, что л. м. с. п. всегда является соседней парой. В последовательности листьев соседняя пара есть л. м. с. п. тогда и только тогда, когда

$$w_{j-2} > w_j \text{ и } w_{j-1} \leq w_{j+1}$$

Пусть $w_1, w_2, \dots, w_{j-1}, w_j, w_{j+1}, \dots, w_n$ — последовательность весов. Опишем для неё алгоритм Гарсиа—Уочса.

Найти самую левую минимальную соседнюю пару, (w_{j-1}, w_j) .

Скомбинировать w_{j-1} и w_j в один узел с весом $w_j^* = w_{j-1} + w_j$.

Передвинуть w_j^* влево, пропуская все узлы, вес которых меньше или равен w_j^* . Получить новую рабочую последовательность из $n - 1$ узла

$$w_1, w_2, \dots, w_i, w_j^*, w_{i+1}, \dots, w_{j-2}, w_{j+1}, \dots, w_n$$

Где $w_i > w_j^* \geq \max\{w_{i+1}, \dots, w_{j-2}\}$.

Этот процесс повторяется, пока в последовательности узлов не останется один узел. Тем самым будет построено дерево из первого шага алгоритма Ху—Таккера. Остальная часть такая же, как в алгоритме Ху—Таккера.

Реализация алгоритмов

Оба алгоритма были реализованы на языке C++ с использованием среды разработки Microsoft Visual Studio, системы контроля версий Git, сервиса GitHub и приложения GitKraken.

1. Структуры

`node` — узел дерева. Содержит информацию о кодируемом знаке (если узел конечный), его весе, уровне в дереве и «детях» (если узел родительский).

```
struct node {  
    size_t id;  
    char sign;  
    size_t weight, level;  
    node* left;  
    node* right;  
};
```

Узлы, которые мы подаём алгоритму на вход, хранятся в памяти программы в экземпляре класса `vector<node>`. Экземпляр `t_nodes` инициализируется с указателями на эти узлы.

Получаемые в ходе выполнения алгоритмов родительские элементы записываются в динамическую область памяти, указатели на них — в общий вектор указателей. Хранение родительских элементов не в экземпляре класса `vector<node>` обусловлено особенностями выделения памяти при добавлении новых элементов в вектор.

2. Функции

Поскольку оба алгоритма использую на заключительном шаге один и тот же стековый алгоритм, рассмотрим сначала реализацию первой части алгоритмов — комбинирования и определения уровней.

Для большей наглядности оба исследуемых алгоритма можно использовать вызовом одной функции.

```
node* buildPseudoTree(bool, vector<node*>&, vector<node>&, node*);
```

Функция `buildPseudoTree` принимает булево значение для выбора алгоритма (`false` — Ху—Таккера, `true` — Гарсия—Уочса), ссылку на последовательность узлов, ссылку на сами узлы и указатель на область в памяти, выделенную для записи родительских элементов. Возвращает эта функция указатель на корневой элемент построенного дерева.

В самой функции `buildPseudoTree` нет реализации алгоритмов, она лишь играет роль «обёртки» для функций `buildPseudoTree_HuTucker` и `buildPseudoTree_GarsiaWachs`.

```
node* buildPseudoTree_HuTucker(vector<node*>&, vector<node>&, node*);
```

Функция принимает те же параметры, что и её «обёртка» `buildPseudoTree`.

```
node* buildPseudoTree_HuTucker(vector<node*>& arr, vector<node>& nodes, node* parents) {
    size_t were = nodes.size();
    size_t i, j;
    while (arr.size() > 1)
    {
        for (i = 0; i < arr.size(); i++) //перебор первого члена пары
        {
            vector<node*> compatibleWithI = findAllCompatibles(arr, i);
            j = min_node(compatibleWithI, false);
            vector<node*> compatibleWithJ = findAllCompatibles(arr, j);
            size_t minCompatibleWithJ = min_node(compatibleWithJ, true);

            if ((minCompatibleWithJ == i) && (i != j))
                break;
        }
        parents[were - arr.size()] = makeParent(arr[i], arr[j], true);
        arr[i] = &parents[were - arr.size()];
        erase(arr, j);
    }
    return arr[0];
}
```

Фиксируется i -й элемент последовательности, для него находятся все совместимые элементы при помощи функции `findAllCompatibles`. С помощью функции `min_node` находится j — id минимального по весу из совместимых элементов. Затем для j -го элемента находятся все совместимые элементы, среди них выбирается минимальный. Если минимальный совместимый элемент для j имеет id i , то элементы i и j образуют локальную минимальную совместимую пару (л.м.с.п.). В этом случае на место i -го записывается родительский элемент, который создаётся функцией `makeParent`, а j -й элемент удаляется из последовательности (стирается указатель из `arr`, сдвигаются и перенумеровываются элементы).

Всё перечисленное выше повторяется до тех пор, пока длина последовательности не станет равна 1. После этого функция вернёт

единственный элемент последовательности — ссылку на корневой узел дерева.

```
node* buildPseudoTree_GarsiaWachs(vector<node*> &, vector<node>&, node*);
```

Эта функция реализует комбинирование и расчёт уровней в алгоритме Гарсия—Уочса. На вход подаются элементы, уже описанные выше для «функции-обёртки» `buildPseudoTree`.

```
node* buildPseudoTree_GarsiaWachs(vector<node*>& arr, vector<node>& nodes, node* parents) {
    size_t were = nodes.size();
    size_t i, j;
    while (arr.size() - 1)
    {
        for (i = 0; i < arr.size()-1; i++) //перебор первого члена пары
        {
            vector<node*> compatibleWithI;
            if (i == 0)
                compatibleWithI.push_back(arr[i + 1]);
            else {
                compatibleWithI.push_back(arr[i - 1]);
                compatibleWithI.push_back(arr[i + 1]);
            }
            j = min_node(compatibleWithI, false);
            vector<node*> compatibleWithJ;
            if (j == 0)
                compatibleWithJ.push_back(arr[j + 1]);
            if (j == arr.size() - 1)
                compatibleWithJ.push_back(arr[j - 1]);
            if (j > 0 && j < arr.size() - 1) {
                compatibleWithJ.push_back(arr[j - 1]);
                compatibleWithJ.push_back(arr[j + 1]);
            }
            size_t minCompatibleWithJ = min_node(compatibleWithJ, true);

            if ((minCompatibleWithJ == i) && (i != j))
                break;
        }
        parents[were - arr.size()] = makeParent(arr[i], arr[j], true);
        arr[i] = &parents[were - arr.size()];
        erase(arr, j);
        while (i > 0 && (arr[i - 1]->weight <= arr[i]->weight))
        {
            swap(arr[i - 1], arr[i]);
            --arr[i - 1]->id;
            ++arr[i]->id;
            --i;
        }
    }
    return arr[0];
}
```

Поскольку алгоритм Гарсия—Уочса подразумевает, что л.м.с.п. — всегда соседняя пара, функция работает только с соседними элементами последовательности.

Фиксируется i -й элемент, соседние с ним складываются в `compatibleWithI`, находится минимальный по весу элемент с `id j` с помощью функции `min_node`. Для элемента j таким же образом находится минимальный среди соседних элементов. Если это i , это л.м.с.п. найдена.

Элемент i заменяется на родительский для i и j , элемент j удаляется из последовательности, элементы перенумеровываются.

После этого родительский элемент передвигается к началу последовательности, пока его вес больше веса лежащего слева.

Всё перечисленное выше повторяется до тех пор, пока длина последовательности не станет равна 1. После этого функция вернёт единственный элемент последовательности — ссылку на корневой узел дерева.

Определение уровней в дереве происходит во время построения этого дерева. По умолчанию все узлы инициализируются с полем `level` равным нулю. Функция `makeParent`, которая создаёт родительский элемент для двух других, рекурсивно увеличивает уровни потомков создаваемого узла, таким образом поле `level` структуры `node` соответствует уровню вложенности в построенном дереве.

Стековый алгоритм

Второй этап работы алгоритмов Ху—Таккера и Гарсия—Уочса заключается в построении дерева на основе посчитанных на предыдущем этапе уровней в псевдодереве.

```
node* buildTree(vector<node>&, node*);
```

Функция принимает ссылку на изначальную последовательность узлов (поля `level`, однако, содержат актуальную информацию об уровнях узлов) и указатель на область в памяти для записи родительских узлов.

Возвращает функция указатель на корневой элемент построенного дерева.

```
node* buildTree(vector<node>& nodes, node* parents) {
    stack<node*> st;
    stack<node*> qu;
    for (int i = nodes.size() - 1; i >= 0; i--) //инициализация стека по имени очередь
        qu.push(&nodes[i]);
    size_t parents_i = 0;
    while (Move1(st, qu, parents, parents_i));
    return st.top();
}
```

Стековый алгоритм состоит из трёх шагов, каждому из которых соответствует функция. Для его работы создаётся два экземпляра класса

`stack<node*>`: `st` (стек) и `qu` (очередь). Указатели на все элементы последовательности по одному отправляются в очередь, после чего вызывается функция `Move1`, в цикле, который остановится, когда `Move1` вернёт `false`.

```
bool Move1(stack<node*>&, stack<node*>&, node*, size_t&);
```

Функция принимает ссылки на стек и очередь, а так же указатель на область памяти для записи родительских элементов. Если два верхних элемента стека равны, осуществляется вызов функции `Move3`, иначе, если два верхних элемента стека неравны или стек содержит менее двух элементов, вызывается функция `Move2`.

```
bool Move1(stack<node*> &st, stack<node*> &qu, node*parents, size_t& parents_i)
{
    if (st.size() < 2)
        Move2(st, qu, parents, parents_i);
    else
    {
        node* top1 = st.top();
        st.pop();
        node* top2 = st.top();
        st.push(top1);
        if (top1->level == top2->level)
            return Move3(st, qu, parents, parents_i);
        else
            Move2(st, qu, parents, parents_i);
    }
}
```

```
void Move2(stack<node*>&, stack<node*>&, node*, size_t&);
```

Функция принимает те же параметры, что и `Move1` и перемещает первый элемент из очереди на вершину стека.

```
void Move2(stack<node*> &st, stack<node*> &qu, node*parents, size_t& parents_i)
{
    st.push(qu.top());
    qu.pop();
    Move1(st, qu, parents, parents_i);
}
```

```
bool Move3(stack<node*>&, stack<node*>&, node*, size_t&);
```

После удаления двух верхних элементов из стека с помощью функции `makeParent` из них формируется родительский лист, который затем помещается на вершину стека. Если при этом его уровень не равен нулю, то функция `Move3` возвращает 1 и алгоритм продолжается, иначе алгоритм завершается, возвращая из функции `buildTree` указатель на корень построенного дерева.

```

bool Move3(stack<node*> &st, stack<node*> &qu, node*parents, size_t& parents_i)
{
    //строим дерево по принципу: два верхних элемента стека st располагаем на
    //соответствующих уровнях
    //помещаем отца на уровень детей - 1, делаем связи между ними
    node* top1 = st.top();
    st.pop();
    node* top2 = st.top();
    st.pop();
    parents[parents_i] = makeParent(top2, top1, false);
    st.push(&parents[parents_i]);
    ++parents_i;
    if (st.top()->level) return 1;
    return 0;
}

```