# Реализация алгоритов 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 и их веса . Эти последовательности назовём соответственно последовательностью узлов и весов.



Если мы комбинируем и , то их отец обозначается через , его вес — через . Если комбинируется и , их отец обозначается через . 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 и , вес их отца называется *весом этой пары*. Пара с минимальным весом называется *минимальной парой*.



Алгоритм Ху—Таккера строит алфавитное дерево, минимизирующее , выполняя следующие шаги.



**Комбинирование.** По данной последовательности из n узлов с весами 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 каждого листа относительно корня (узел с наибольшим значением располагается внизу дерева).



**Перестройка.** После того, как номера уровней всех листьев определены, применим ним стековый алгоритм. Он заключается в следующих шагах.



Шаг 0. Стек пуст, 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 – верхний элемент стека, а — следующий элемент. Заменить и на . Если , остановиться, иначе перейти на шаг 1. (Это означает, что комбинируются узлы и , а их отец становится узлом уровня .)



### 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

и



Пусть — последовательность весов. Опишем для неё алгоритм Гарсиа—Уочса.



Найти самую левую минимальную соседнюю пару, .



Скомбинировать и в один узел с весом .



Передвинуть влево, пропуская все узлы, вес которых меньше или равен . Получить новую рабочую последовательность из n – 1 узла



Где .



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 С++ 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\_nodeнаходится *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;

size\_t max\_level = 0;

for (int i = nodes.size() - 1; i >= 0; i--) //инициализация стека по имени очередь

{

if (nodes[i].level > max\_level)

max\_level = nodes[i].level;

qu.push(&nodes[i]);

}

size\_t parents\_i = 0;

while (Move1(st, qu, parents, parents\_i, max\_level));

return st.top();

}

Для работы стекового алгоритма 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&, size\_t);

Функция принимает ссылки на стек и очередь, указатель на область памяти для записи родительских элементов и параметр max\_level, указывающий на максимальный уровень среди всех узлов в последовательности.

bool Move1(stack<node\*> &st, stack<node\*> &qu, node\*parents, size\_t& parents\_i, size\_t max\_level)

{

if (st.size() < 2)

Move2(st, qu, parents, parents\_i, max\_level);

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, max\_level);

else

Move2(st, qu, parents, parents\_i, max\_level);

}

}