

Московский авиационный институт
(Национальный исследовательский университет)

Факультет «Информационные технологии и прикладная математика»

Курсовая работа по курсу
Дискретный анализ
на тему
«Изоморфизм корневых деревьев»

Студент: Ваньков Д. А.

Группа: М80-207Б-17

Преподаватель: Журавлев А. А.

Дата:

Постановка задачи

Необходимо реализовать программу, принимающую на вход 2 числа: количество деревьев и их размер, а также сами деревья, заданные в виде массива предков. Формат вывода: число групп изоморфных деревьев, а затем размер каждой группы и индексы деревьев, входящих в данную группу. Вывод должен быть отсортирован по размеру групп, и у групп с одинаковым размером по 1 индексу.

Метод решения

Два дерева называются изоморфными, если они имеют одну структуру. К примеру, два дерева «А» и «В» будут изоморфны друг другу:



Пусть каждое дерево имеет свой код, который создается следующим образом: если вершина является листом - ей присваивается код «10», а если нет, то ее кодом будет являться конкатенация кодов ее детей и приписанные в начало «1» и «0» в конец. Таким образом, пусть у вершины k есть два ребенка с кодами $str1$ и $str2$ соответственно, тогда код вершины k будет выглядеть следующим образом: $k = 1 + str1 + str2 + 0$. Получается код всего дерева будет содержаться в его корне. Можно заметить, что у двух изоморфных деревьев будет одинаковый код, если его отсортировать. Отсортированы кодом назовем код, полученный поэтапной сортировкой кодов, расположенных от листов к корню. Т.е. к примеру, мы имеем 3 кода: 1100, 10 и 110100, тогда в код текущей вершины сначала запишется 10, затем 1100 и уже после 110100 и мы получим: $code = 1 + 10 + 1100 + 110100 + 0$.

Обход и получение кода

Первоначально, на вход подается только массив предков каждой вершины, то есть в i -ом индексе содержится число (номер вершины), являющееся предком вершины с номером i . Чтобы получить список смежности вершин нужно просто один раз пройти по массиву и заполнить для каждой вершины массив вершин, который по сути является массивом детей этой вершины.

Для хранения дерева создадим структуру Rtree, в которой будет находиться вектор — само дерево, строка — его код, индекс дерева, список смежности для каждой из вершин и его код.

Для обхода деревьев и получения их кодов будем использовать поиск в глубину. Как только мы приходим в лист, присваиваем ему код «10», и возвращаем этот код как результат функции. Тогда для k — ой вершины мы получим, после обхода n ее детей — n кодов, которые нужно просто отсортировать и приписать «1» в начало и «0» в конец. Таким образом мы получили коды всех деревьев, которые осталось только сравнить. Для этого лучше всего подойдет структура `std::map`, ключом у которой будет являться сам массив предков, а значением строка — код дерева. Заведем вектор результата, в котором будет содержаться все группы изоморфных деревьев. С помощью итераторов пройдемся по мапе и, если значения совпали, запомним индекс данного дерева. То есть после прохождения по мапе мы получим в векторе результата n векторов — индексов изоморфных деревьев, где n — количество изоморфных групп.

Для получения ответа осталось только отсортировать вектор по размеру групп и по первому индексу в группах с одинаковым размером. Для этого можно написать простой компаратор и воспользоваться сортировкой `std::sort`.

Данный алгоритм носит название АНУ, в честь трёх ученых Aho, Hopcroft и Ullman. Работает за время $O(n \cdot \log n)$, так как получение списка смежности занимает $O(n)$, обход в глубину $O(n)$, и сортировка $O(n \cdot \log n)$.

Список литературы

1. https://logic.pdmi.ras.ru/~smal/files/smal_jass08_slides.pdf

Листинг и описание программы

main.cpp

```
#include <iostream>
#include <string>
#include <map>
#include <vector>
#include <algorithm>

struct Rtree {
    std::vector<int> parent;
    std::vector<std::vector<int>>> Struct_Adj;
    std::string tree_code;
    int index;
};
```

```

void Read(std::vector<Rtree> &Data) {
    for (int i = 0; i < Data.size() - 1; ++i) {
        for (int j = 0; j < Data[i].parent.size(); ++j) {
            std::cin >> Data[i].parent[j];
        }
        Data[i].index = i;
    }
}

void Print(std::vector<Rtree> &Data) {
    for (int i = 0; i < Data.size() - 1; ++i) {
        std::cout << "ind " << Data[i].index << " value ";
        for (int j = 0; j < Data[i].parent.size(); ++j) {
            std::cout << Data[i].parent[j] << " ";
        }
        std::cout << "\n";
    }
}

bool cmp(const std::string &lhs, const std::string &rhs) {
    return lhs < rhs;
}

std::vector<std::string> new_sort_str(std::vector<std::string> &vector) {
    std::sort(vector.begin(), vector.end(), cmp);
    return vector;
}

std::string get_full_str(std::vector<std::string> &vector) {
    vector = new_sort_str(vector);
    std::string result;
    for (int i = 0; i < vector.size(); ++i) {
        result += vector[i];
    }
    return result;
}

std::string str_dfs(int vertex, std::vector<Rtree> &Data, std::vector<char> &used, int k, std::string &code) {
    used[vertex] = true;
    std::vector<std::string> tmp;
    std::string str;

    //std:: cout << vertex << "\n";
    if (Data[k].Struct_Adj[vertex].empty()) {
        return "10";
    }

    for (std::vector<int>::iterator i = Data[k].Struct_Adj[vertex].begin();
        i != Data[k].Struct_Adj[vertex].end(); ++i) {
        if (!used[*i]) {
            //tmp += str_dfs(*i, Data, used, k, code);
            //tmp = sort_str(tmp, str_dfs(*i, Data, used, k, code));
            tmp.push_back(str_dfs(*i, Data, used, k, code));
            str = get_full_str(tmp);
        }
    }
}

```

```

    }

    code = '1' + str + '0';
    return code;
}

void Tree_str_dfs(std::vector<Rtree> &Data) {
    for (int k = 0; k < Data.size() - 1; ++k) {
        std::string code;
        std::vector<char> used;
        used.resize(Data[k].parent.size() + 1);
        code = str_dfs(Data[k].parent[0], Data, used, k, code);
        Data[k].tree_code = code;
        //std::cout << "index: " << Data[k].index << " code: " << code <<
"\n";
    }

}

bool res_cmp(const std::vector<int> &lhs, const std::vector<int> &rhs) {
    return lhs.size() < rhs.size();
}

bool first_ind_cmp(const std::vector<int> &lhs, const std::vector<int>
&rhs) {
    if (lhs.size() == rhs.size())
        return lhs[0] < rhs[0];
    else
        return false;
}

void AHU(std::vector<Rtree> &Data, int n) {
    std::vector<std::vector<int>> Adj;

    //Получение массива детей (список смежности)
    for (int i = 0; i < Data.size() - 1; ++i) {
        Data[i].Struct_Adj.resize(n + 1);
        for (int j = 0; j < Data[i].parent.size(); ++j) {
            Data[i].Struct_Adj[Data[i].parent[j]].push_back(j + 1);
        }
    }

    Tree_str_dfs(Data);

    std::map<std::string, std::vector<int>> mapp;
    std::vector<bool> was_taken(Data.size() - 1, false);
    for (int i = 0; i < Data.size() - 1; ++i) {
        if (was_taken[i] == false) {
            mapp[Data[i].tree_code].push_back(Data[i].index);
            was_taken[i] = true;
        }
        for (int j = i + 1; j < Data.size() - 1; ++j) {
            if (Data[i].tree_code == Data[j].tree_code && was_taken[j] ==
false) {
                was_taken[j] = true;
                mapp[Data[j].tree_code].push_back(Data[j].index);
            }
        }
    }
}

```

```

std::vector<std::vector<int>> result;
std::cout << mapp.size() << "\n";
for (auto &i: mapp) {
    result.push_back(i.second);
}

std::sort(result.begin(), result.end(), res_cmp);
std::sort(result.begin(), result.end(), first_ind_cmp);
for (int i = 0; i < result.size(); ++i) {
    std::cout << result[i].size() << " ";
    for (int j = 0; j < result[i].size(); ++j) {
        std::cout << result[i][j] << " ";
    }
    std::cout << "\n";
}

//Вывод списка смежности
/*for (int i = 0; i < Data.size() - 1; ++i) {
    std::cout << "Index_ " << Data[i].index << "\n";
    for (int k = 0; k < Data[i].Struct_Adj.size() - 1; ++k) {
        std::cout << k << ": ";
        for (int j = 0; j < Data[i].Struct_Adj[k].size(); ++j) {
            std::cout << Data[i].Struct_Adj[k][j] << " ";
        }
        std::cout << "\n";
    }
}*/
}

int main() {
    std::vector<Rtree> Data;
    std::vector<int> tree;
    int k, n, N = 0;
    std::cin >> k >> N;
    n = N - 1;
    if (k == 0 || n == 0 || n == -1) {
        std::cout << "0\n";
        return 0;
    }
    Data.resize(k + 1);
    for (int i = 0; i < Data.size(); ++i) {
        Data[i].parent.resize(n);
    }

    Read(Data);
    //Print(Data);
    AHU(Data, n);
    return 0;
}

```

Makefile

```

CC = g++
CFLAGS = -std=c++11 -O2 -pedantic -Wall -Werror -Wno-sign-compare

all:
    $(CC) $(CFLAGS) main.cpp -o main

clear:

```

```
rm *.o main
```

Тесты

```
chappybunny@chappybunny:~/CLionProjects/KP_tree$ cat test.txt
7 5
0 0 0 1
0 0 0 2
0 0 1 1
0 0 2 2
0 1 1 3
0 1 2 3
0 1 1 3
chappybunny@chappybunny:~/CLionProjects/KP_tree$ time ./main < test.txt
4
1 5
2 0 1
2 2 3
2 4 6

real    0m0,004s
user    0m0,001s
sys     0m0,004s
```

```
chappybunny@chappybunny:~/CLionProjects/KP_tree$ cat test.txt
10 11
0 0 0 1 1 1 2 2 3 3
0 0 0 1 1 2 2 2 3 3
0 0 0 1 1 2 2 3 3 3
0 0 0 1 1 1 2 2 7 8
0 0 0 2 2 3 3 3 4 5
0 0 0 2 2 2 3 3 7 8
0 0 0 1 1 1 3 3 7 8
0 0 0 1 1 3 3 3 4 5
0 0 0 1 1 2 2 2 4 5
0 0 0 1 4 4 4 7 7 7
chappybunny@chappybunny:~/CLionProjects/KP_tree$ time ./main < test.txt
3
1 9
3 0 1 2
6 3 4 5 6 7 8

real    0m0,006s
user    0m0,001s
sys     0m0,005s
```

Вывод

Применение

На практике необходимость проверки изоморфизма деревьев возникает при решении задач хемоинформатики, математической или компьютерной химии, автоматизации проектирования электронных схем (верификация различных представлений электронной схемы) и оптимизации программ (выделение общих подвыражений). Последнее представляет собой наиболее интересное

применение. Представим, что студент при написании курсовой работы решил списать код из интернета, поменяв при этом только название переменных и функции, и оставив алгоритм не именованным. Тогда, так как компилятор строит дерево решения, дерево, полученное от первоначального кода будет изоморфно дереву исправленного кода. Таким образом можно выяснить, что студент списал.

Эта работа была интересной для меня. Я подробнее познакомился с структурами `std::map` и `std::vector`, и узнал как можно отсортировать вектор с любыми данными внутри, используя `std::sort`, написав свой компаратор.

На этапе сборке, чтобы ускорить работу программы я использовал ключ `-O2`, который увеличивает время компиляции, однако ускоряет ее запуск.