Санкт-Петербургский политехнический университет Петра Великого
Институт металлургии, машиностроения и транспорта
Высшая школа автоматизации и робототехники

Курсовая работа

Дисциплина: Программирование на языках высокого уровня

Тема: Алгоритм Тарьяна

Выполнил
студент гр. 3331506/70401 Засецкий В.С.
Преподаватель Ананьевский М.С.

« »_____2020 г.

Санкт-Петербург
2020

Оглавление

1. Введение

Алгоритм Тарьяна — алгоритм поиска компонент сильной связности в орграфе. На вход алгоритму подаётся ориентированный граф, на выходе получаются вершины, разбитые на компоненты сильной связности.

2. Словесное описание алгоритма

Для реализации алгоритма необходимо иметь стек, куда будут заноситься встреченные узлы и 2 массива, индексированных номерами узлов: массив *id* с порядковым номером узла и массив *low_link_value* с наименьшим номером связи узла (наименьший индекс, до которого может добраться узел при поиске в глубину). Также потребуется булев массив *is_used* размером с количество узлов, который будет сообщать о том, запускалась ли функция поиска в глубину из этого узла.

Поиск в глубину начинается с произвольного начального узла. Поиск посещает каждую вершину графа только единожды, не заходя в ранее рассмотренные вершины. Последующие поиски в глубину начинаются с ещё не рассмотренных вершин. При попытке захода в уже рассмотренный узел или при возврате рекурсии из узла-потомка в текущий узел, если узел-потомок находится в стеке, происходит присвоение *low_link_value* текущего узла минимального значения min(*low_link_value[current_node], low_link_value[child]*). Если при возврате рекурсии значения *low_link_value* и *id* узла совпадут, значит данный узел является «корнем» компоненты сильной связности. В таком случае вынимаем из стека и выводим как элементы компоненты сильной связности все узлы вплоть до узла-«корня».

3. Реализация алгоритма

Для реализации алгоритма был создан класс Graph. Класс имеет атрибуты: количество узлов *amount_of_nodes,* массив двусвязных списков с соединениями для каждого узла *node_connections*, счётчик порядкового номера узла

node_number_counter, а также структуру *param* с массивами *id* и *low_link_value*, булевыми массивами *is_in_stack* и *is_used* и стеком. Методы класса: добавление соединения *add_connection* и поиск компонент сильной связности *find_SCC*.

Программный код с описанием класса представлен ниже.

```
typedef unsigned int uint;

struct parameters
{
    uint * node_id;
    uint * low_link_value;
    std::stack<uint> stack;
    bool * is_in_stack;
    bool * is_used;
};

class Graph
{
private:
    uint amount_of_nodes;
    std::list<uint> * node_connections;
    uint node_number_counter;
    parameters param;
public:
    Graph(uint size)
    {
        amount_of_nodes = size;
        node_number_counter = 0;
        node_connections = new std::list<uint>[size];
        param.node_id = new uint[size];
        param.low_link_value = new uint[size];
        param.is_in_stack = new bool[size];
        param.is_used = new bool[size];
    }
    ~Graph() {}
    void add_connection(uint from, uint to);
    void find_SCC();
    void set_random_connections();

private:
    void dfs(uint current_node);
};
```

Программный код функции поиска компонент сильной связности, а также функции добавления соединения приведён ниже.

```
void Graph::add_connection(uint from, uint to)
{
    if (from >= amount_of_nodes || to >= amount_of_nodes)
    {
        std::cout << "Node number excess" << std::endl;
        return;
    }
    if (from == to)
    {
        std::cout << "\"from\" and \"to\" must be different nodes" << std::endl;
```

```
    }
    node_connections[from].push_back(to);
    return;
}


void Graph::find_SCC()
{
    for (int j = 0; j < this->amount_of_nodes; j++)
    {
        param.is_in_stack[j] = false;
        param.is_used[j] = false;
    }

    for (int node = 0; node < this->amount_of_nodes; node++)
    {
        if (param.is_used[node] == false)
            dfs(node);
    }
}



void Graph::dfs(uint current_node)
{
    //присваиваем id узлу
    param.node_id[current_node] = this->node_number_counter;
    this->node_number_counter++;
    //наименьший номер связи равен id
    param.low_link_value[current_node] = param.node_id[current_node];
    //кладём узел в стек
    param.stack.push(current_node);
    param.is_in_stack[current_node] = true;
    param.is_used[current_node] = true;

    //перебираем соединения узла
    std::list<uint>::iterator connections_iterator;
    for (connections_iterator = this->node_connections[current_node].begin();
         connections_iterator != this->node_connections[current_node].end();
         connections_iterator++)
    {
        uint current_connection = * connections_iterator;
        //если узел ещё не был использован, вызываем рекурсивную функцию для него
        if (param.is_used[current_connection] == false)
        {
            dfs(current_connection);
            //после возврата из рекурсии определяем наименьший номер связи текущего
узла
            if (param.is_in_stack[current_connection])
                param.low_link_value[current_node] =
std::min(param.low_link_value[current_node], param.low_link_value[current_connection]);
        }
        //если узел уже был использован
        else
        {
            //если узел в стеке, определяем наименьший номер связи текущего узла
            if (param.is_in_stack[current_connection] == true)
                param.low_link_value[current_node] =
std::min(param.low_link_value[current_node], param.low_link_value[current_connection]);
        }
    }
```

```
    //когда прошлись по всем связям узла, проверяем, совпадает ли наименьший номер
связи и id узла
    if (param.node_id[current_node] == param.low_link_value[current_node])
    {
        //если совпадает, значит этот узел - "корень" компоненты сильной связности
(КСС)
        //выводим КСС на экран
        std::cout << "SCC: ";
        uint top_node;
        while (param.stack.top() != current_node)
        {
            top_node = param.stack.top();
            std::cout << top_node << " ";
            param.stack.pop();
            param.is_in_stack[top_node] = false;
        }
        top_node = param.stack.top();
        param.stack.pop();
        param.is_in_stack[top_node] = false;
        std::cout << top_node << std::endl;
    }
}
```

4. Анализ алгоритма

Временная сложность

Процедура поиска в глубину вызывается единожды для каждого узла, а количество проверок смежных узлов на возможность вызова поиска в глубину равно количеству рёбер. Соответственно, сложность алгоритма Тарьяна линейна: O(m+n), где n – число вершин, m – число ребер.

Время выполнения алгоритма в зависимости от количества узлов в графе:

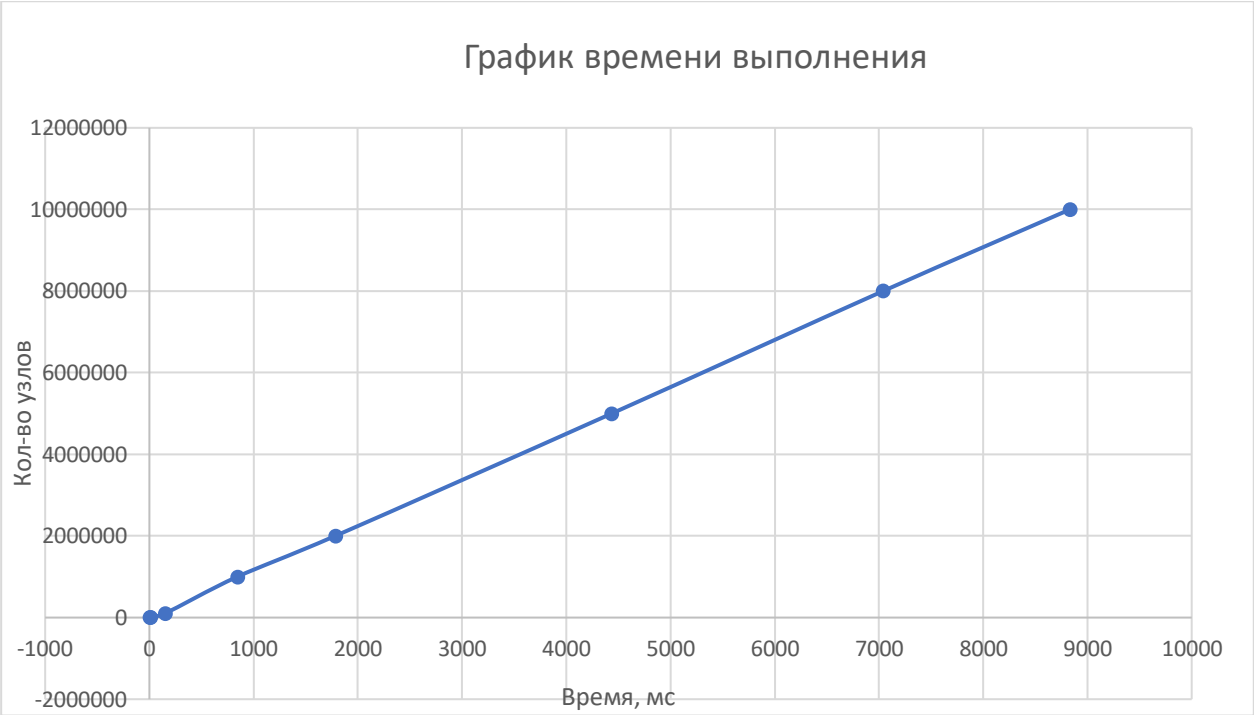

Рисунок 1 — Время выполнения алгоритма

5. Применение алгоритма

Алгоритм Тарьяна может применяться для топологической сортировки узлов орграфа, не содержащего контуров. Также этот алгоритм может применяться в качестве начального для алгоритмов, работающих только для компонент сильной связности. Алгоритм Тарьяна может быть применён для решения “2-satisfiability” проблемы, для расчёта декомпозиции Далмейджа-Мендельсона.

Заключение

В работе был рассмотрен алгоритм Тарьяна и его реализация на языке C++. Достоинствами алгоритма Тарьяна можно назвать линейное время выполнения и использование одного стека вместо двух, как в других алгоритмах поиска компонент сильной связности.

Список литературы

1. *Роберт Седжвик.* Глава 5. Метод уменьшения размера задачи: Топологическая сортировка // Алгоритмы на графах = Graph algorithms. — 3-е изд. — Россия, Санкт-Петербург: «ДиаСофтЮП», 2002. — С. 496. — ISBN 5-93772-054-7.
2. *Кормен, Т., Лейзерсон, Ч., Ривест, Р., Штайн, К.* Глава 23.1.3. Поиск в глубину // Алгоритмы: построение и анализ / Под ред. И. В. Красикова. — 2-е изд. — М.: Вильямс, 2005. — С. 632-635. — ISBN 5-8459-0857-4.