ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»
Институт машиностроения, материалов и транспорта
Высшая школа автоматизации и робототехники

КУРСОВАЯ РАБОТА

Алгоритм Косарайю

по дисциплине «Объектно-ориентированное программирование»

Выполнил: студент гр. 3331506/90401 Скрыль М.И.
Преподаватель: Ананьевский М.С.

Санкт-Петербург
2022

Введение

Алгоритм Косарайю является алгоритмом поиска областей сильной связности в ориентированном графе.

Две вершины (u и v) ориентированного графа называют сильно связными, если существует путь из u в v и существует путь из v в u. Ориентированный граф называется сильно связным, если *любые две* его вершины сильно связны.

Компоненты сильной связности — такое разбиение вершин, что внутри одной компоненты все вершины сильно связаны, а между вершинами разных компонент сильной связности нет. Областью сильной связности называется множество вершин компонентов сильной связности.

Поиск компонент сильной связности (следовательно, и алгоритм Косарайю) применяют для последующей конденсации графа, то есть сжатия каждой компоненты сильной связности в одну вершину, который в свою очередь используется для решения задачи 2-SAT (или задача выполнимости булевых формул), которая формулируется следующим образом. Рассмотрим, например, такое выражение: $(a \mid b)$ & $(!\, c \mid d)$ & $(!\, a \mid !\, b)$. Задача выполнимости (SAT) заключается в том, чтобы найти такие значения переменных, при которых выражение становится истинным, или сказать, что такого набора значений нет. Для примера выше такими значениями являются $a = 1, b = 0, c = 0, d = 1$.

Известно, что выражение $a \mid b$ эквивалентно $(!\, a \to b) | \, (!\, b \to a)$, где $\to$ означает импликацию, следовательно, можно составить граф импликаций. Если какое-то выражение положить верным, то из него будут достижимы все остальные выражения, которые следуют из него. Но, если для какой-то переменной выполняется, что из x достижимо $!\, x$, а из $!\, x$ достижимо x, то задача решения не имеет, так как в таком случае, какое бы значение x ни было выбрано, всегда будет получено противоречие — что должно быть выбрано и обратное ему значение.

С точки зрения теории графов это означает, что из первой вершины достижима вторая и, наоборот, из второй – первая, то есть эти вершины находятся в одной компоненте сильной связности. Тогда условие существования

решения звучит так: для того, чтобы задача 2-SAT имела решение, необходимо, чтобы для любой переменной x вершины x и $!x$ находились в разных компонентах сильной связности графа импликаций.

С SAT же связано большое количество практических задач, таких как:

1) Многие рекомендательные системы (например, в составе Netflix) используют алгоритм разложения булевых матриц для рекомендации контента. Оптимальное разложение таких матриц может быть найдено с помощью SAT.

2) Задача оптимального распределения задач по процессорам

3) Популярный метод кластеризации k-means (метод к-средних)

4) Задачи о сериализации истории транзакций в базах данных

Описание алгоритма

Перед объяснением самого алгоритма обратимся к основным понятиям, которые важны для его понимания.

Основу алгоритма составляет поиск в глубину или depth-first search (DFS), общая идея которого состоит в следующем: для каждой непройденной вершины необходимо найти все непройденные смежные вершины и повторить поиск для них.

Такт DFS из вершины v – обход (в глубину) всех вершин графа, достижимых из v. Такт можно интерпретировать как рекурсивный вызов функции. Такт обработки вершины, у которой нет соседей, будет равняться 1.

Инвертирование графа – смена направлений всех рёбер в графе на противоположные.

Время выхода вершины – число, соответствующее времени выхода рекурсии алгоритма DFS из вершины. Притом, изначально счётчик времени 0, увеличивается он лишь в двух случаях:

1. Начало нового такта DFS

2. Прохождение по ребру (при том, не важно, рекурсивный проход или нет)

Алгоритм Косарайю состоит из трех шагов:

1. Выполняется поиск в глубину, пока всем вершинам не будет присвоено время выхода из рекурсии.

2. Исходный граф инвертируется

3. Еще раз выполняется поиск в глубину, начиная с вершины с наибольшим временем выхода.

Вершины, посещенные за один такт DFS в третьем пункте, будут являться компонентами сильной связности.

Пример:

Допустим, что дан следующий граф:

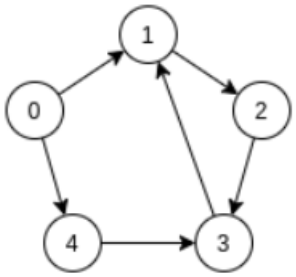

Рисунок 1 – Пример

Запускаем DFS из вершины «0».

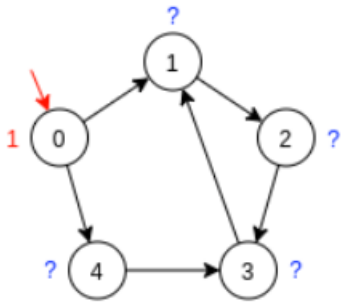

Рисунок 2 – Пример

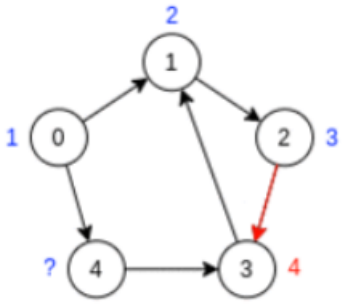

Рисунок 3 – Пример

Доходим до вершины 3, у которого нет непосещенных смежных вершин, начинаем возвращаться назад. Так как у вершин 2 и 1 все смежные вершины тоже посещены, возвращаемся в вершину 0.

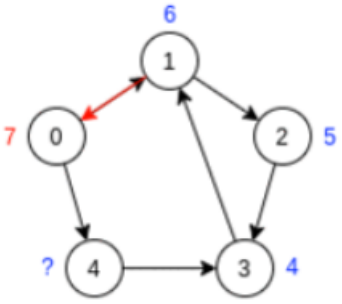

Рисунок 4 – Пример

Посещаем вершину 4.

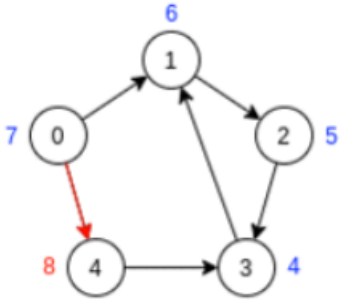

Рисунок 5 – Пример

Так как у нее больше нет непосещенных смежных вершин, возвращаемся в вершину 0.

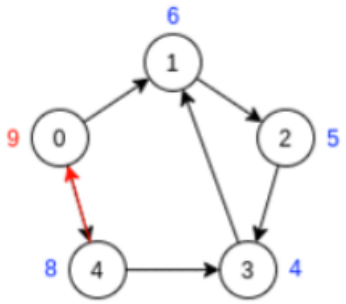

Рисунок 6 – Завершение DFS

У вершины 0 больше нет непосещенных вершин, к тому же все вершины уже посещены и отмечены, DFS завершается.

Далее инвертируем ребра исходного графа.

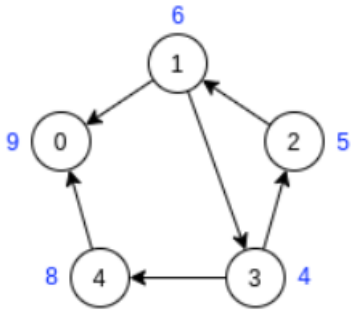

Рисунок 7 – Инвертированный граф

Запускаем DFS по инвертированному графу из вершины 0, так как у нее наибольшее время выхода.

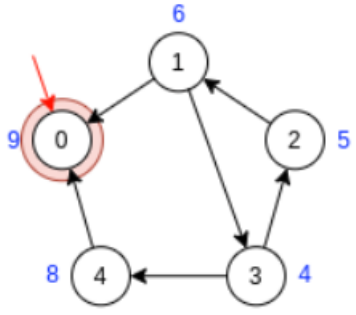

Рисунок 8 – Второй DFS

Так как из нее нельзя посетить ни одной смежной вершины, такт DFS заканчивается, следующий начинается из вершины 4, а вершина 0 является компонентой сильной связности.

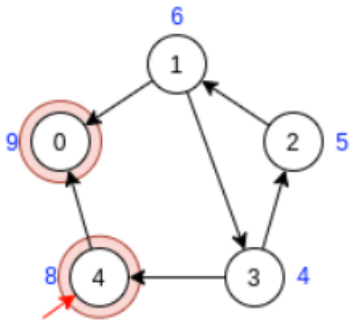

Рисунок 9 – Пример

Из вершины 4 есть возможность попасть в вершину 0, но, так как она уже посещена, то такт DFS заканчивается, новый начинается с вершины 1, а вершина 4 является компонентой сильной связности.

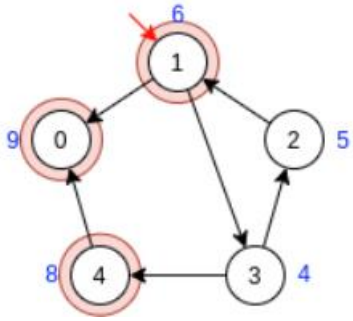

Рисунок 10 – Пример

Вершина 3 является непосещенной, так что DFS из вершины 1 переходит в вершину 3, а затем в вершину 5, у которой уже не будет непосещенных смежных вершин, следовательно, такт DFS завершится, а вершины 1-3-2 будут являться компонентой сильной связности.

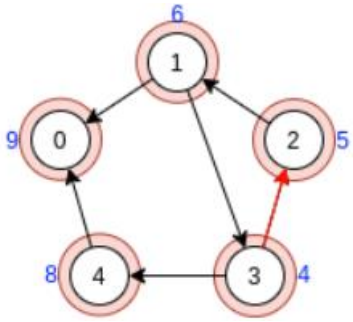

Рисунок 11 – Алгоритм завершен

После завершения алгоритма получим три компоненты сильной связности.

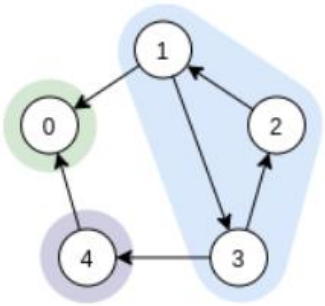

Рисунок 12 – Компоненты сильной связности

Исследование алгоритма

Этот алгоритм состоит из двух поисков в глубину, которые и определяют его временную сложность, которая составляет $O(V^2)$ в случае насыщенных графов и $O(V+E)$ в случае разреженных. Для исследования времени выполнения алгоритма было создано большое количество случайных графов с количеством ребер, соответствующим худшему случаю алгоритма, для которых было замерено время выполнения алгоритма. График представлен на рисунке 13.

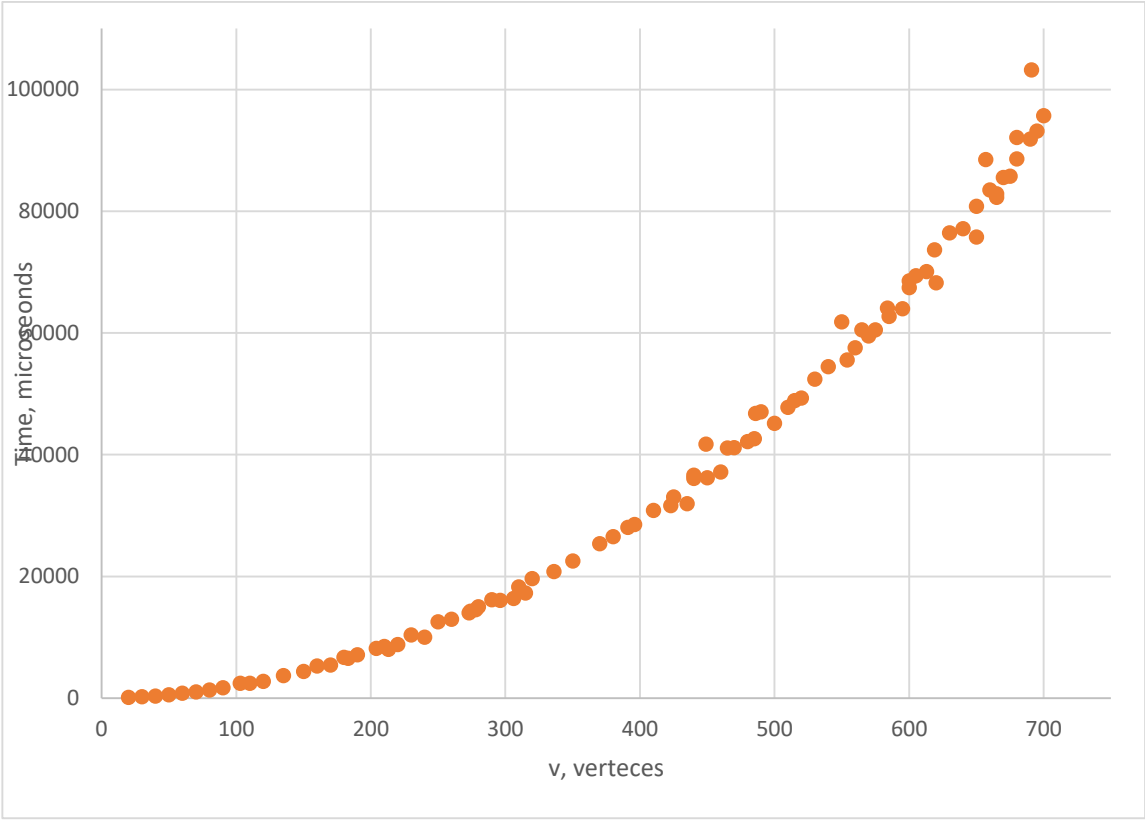

Рисунок 13 – Зависимость временной алгоритма от количества вершин в худшем случае

Заключение

В работе был рассмотрен алгоритм Косайрайю для нахождения сильно связанных компонент в ориентированном графе. Было выявлено, что для разреженных графов временная сложность алгоритма является линейной зависимостью, а для плотных графов – квадратичной, что делает его менее эффективным при поиске компонент сильной связности в плотных графах по сравнению с алгоритмом Тарьяна, обладающего временной сложностью $O(V + E)$, тем не менее алгоритм Косарайю является, возможно, наиболее концептуально понятным из всех алгоритмов, выполняющих поиск компонент сильной связности в ориентированном графе.

Список литературы

1. Роберт Седжвик, Фундаментальные алгоритмы на C++. Часть 5: алгоритмы на графах. – 3-е изд. – СПб: ООО «ДиаСофтЮП», 2002. – 496 с.

2. John Carey. C++ Data Structures and Algorithm Design Principles – UK, Birmingham, Packt Publishing Ltd., 2019. - 626 с.

3. Алгоритм Kosaraju по полкам [Электронный ресурс] // https://habr.com/ru/post/537290/

Приложение

```cpp
#include <iostream>
#include <vector>
#include <chrono>
#include <algorithm>
using namespace std;

class Graph {
private:
    int vertices;
    int tout_counter = 0;
    vector<vector<int>> adj_list;
    vector<vector<int>> reversed;
    vector<bool> visited;
    vector<int> tout;
public:
    explicit Graph(int v);
    void adjacency_list(int u, int v);
    void DFS_recursive();
    bool is_connected(int u, int v);
public:
    void print_list();
private:
    int max_tout();
    void DFS(int u);
    void DFS_reversed(int u);
};
```

```cpp
Graph::Graph(int v) {
    vertices = v;
    for (int i = 0; i < vertices; ++i) {
        visited.push_back(false);
        tout.push_back(0);
        adj_list.emplace_back();
        reversed.emplace_back();
    }
}

void Graph::adjacency_list(int u, int v) {
    adj_list[u].push_back(v);
    reversed[v].push_back(u);
}

void Graph::DFS_recursive() {
    for (int i = 0; i < vertices; ++i) {
        if (!visited[i]) {
            DFS(i);
        }
    }
    visited.assign(vertices,  x: false);
    for (auto const & i: visited) {
        if (!i) {
            DFS_reversed( u: max_tout());
            cout << endl;
        }
    }
}
```

```cpp
void Graph::DFS(int u) {
    tout_counter++;
    visited[u] = true;
    for (auto & i: adj_list[u]) {
        if (!visited[i]) {
            DFS(i);
            tout_counter++;
        }
    }
    tout[u] = tout_counter;
}

int Graph::max_tout() {
    int max = 0;
    int max_vert = 0;
    for (int i = 0; i < tout.size(); ++i) {
        if (tout[i] > max) {
            max = tout[i];
            max_vert = i;
        }
    }
    tout.erase( position: tout.begin() + max_vert);
    tout.insert( position: tout.begin() + max_vert,  x: 0);
    return max_vert;
}
```

```
void Graph::DFS_reversed(int u) {
    visited[u] = true;
    cout << u << "  ";
    for (auto &i: reversed[u]) {
        if (!visited[i]) {
            tout.erase( position: tout.begin() + i);
            tout.insert( position: tout.begin() + i,  x: 0);
            DFS_reversed(i);
        }
    }
}

bool Graph::is_connected(int u, int v) {
    if (!adj_list[u].empty()) {
        if (find( first: adj_list[u].begin(),  last: adj_list[u].end(), v) != adj_list[u].end()) {
            return true;
        } else {
            return false;
        }
    } else {
        return false;
    }
}
```

```
void rand_list() {
    srand( seed: time( _timer: nullptr));
    int vertices = rand() % 500;
    Graph g(vertices);
    int edges = rand() % ((vertices) * (vertices - 1));
    cout << endl << "V + E = " << edges + vertices << endl;
    for (int i = 0; i < edges; ++i) {
        int u = 0;
        int v = 0;
        do {
            u = rand() % vertices;
            v = rand() % vertices;
        } while ((u >= vertices) || (v >= vertices) || (u == v) || (g.is_connected(u,v)));
        g.adjacency_list(u, v);
    }
    g.print_list();
    g.DFS_recursive();
}

void Graph::print_list() {
    for (int i = 0; i < vertices; ++i) {
        cout << i << "->";
        for (auto j: adj_list[i]) {
            cout << j << "->";
        }
        cout << endl;
    }
}
```

```
int main() {
    //rand_list();
    Graph gr( v: 8);
    gr.adjacency_list( u: 0,  v: 1);
    gr.adjacency_list( u: 1,  v: 2);
    gr.adjacency_list( u: 2,  v: 3);
    gr.adjacency_list( u: 3,  v: 0);
    gr.adjacency_list( u: 2,  v: 4);
    gr.adjacency_list( u: 4,  v: 5);
    gr.adjacency_list( u: 5,  v: 6);
    gr.adjacency_list( u: 6,  v: 4);
    gr.adjacency_list( u: 6,  v: 7);
    gr.DFS_recursive();
    return 0;
}
```