Санкт-Петербургский политехнический университет Петра Великого

Институт машиностроения, материалов и транспорта

Высшая школа автоматизации и робототехники

Курсовая работа

Дисциплина: Объектно-ориентированное программирование

Тема: Алгоритм Тарьяна для нахождения мостов

Студент гр. 3331506/90401 Черников А. В.

Преподаватель Ананьевский А. С.

«____» ________ 2022 г.

Санкт-Петербург

2022 г.

Оглавление

Введение

Граф — это множество точек (вершин, узлов), которые соединяются множеством линий (рёбер, дуг). Неориентированный граф – граф, рёбра которого не имеют определённого направления. Связный граф – граф, который содержит одну компоненту связности.

В некоторых ситуациях, когда важно, чтобы граф был связным, может оказаться существенным тот факт, что он остается связным, если убрать из него какую-либо вершину или ребро. То есть, мы, возможно, захотим знать более одного пути между каждой парой вершин графа с тем, чтобы застраховаться от возможных отказов и неисправностей.

Например, мы можем лететь из Нью-Йорка в Сан-Франциско, даже если аэропорт в Чикаго завален снегом, ибо существует рейс через Денвер. Или можно вообразить себе ситуацию во время военных действий, в условиях которой мы хотим проложить такую железнодорожную сеть, когда противник, дабы нарушить железнодорожное сообщение, должен разбомбить, по меньшей мере, две станции. Аналогично, мы вправе рассчитывать, что соединения в интегральной схеме или в сети связи проложены таким образом, что остальная часть схемы продолжает работать, если оборвался какой-либо провод или какое-то соединение перестало работать.

Ребро в неориентированном связном графе является мостом, если его удаление разъединяет граф. Для несвязанного неориентированного графа определение аналогично, мост – это ребро, удаление которого увеличивает число компонентов связности. Подобно точкам сочленения, мосты представляют собой уязвимости в подключенной сети и полезны для проектирования надежных сетей.

Описание алгоритма

Простой подход заключается в том, чтобы один за другим удалить все ребра и посмотреть, не приведет ли удаление ребра к отключению графика. Временная сложность этого метода составляет $O\,(E \cdot (V + E))$, где E – количество ребер, V – количество вершин.

Алгоритм Тарьяна для поиска мостов основан на обходе графа в глубину (или *depth-find search*, сокращенно *DFS*) и имеет сложность $O\ (V\ +\ E)$.

В этом алгоритме используется следующее свойство: в любом дереве *DFS* ребро *v-w* есть мост тогда и только тогда, когда не существует обратное ребро, которое соединяет один из потомков *w* с каким-либо предком *v*.

Провозглашение этого свойства эквивалентно утверждению, что единственная связь поддерева с корнем в *w*, ведущая в узел, который не входит в это поддерево, есть родительская связь, ведущая из *w* назад в *v*. Это условие соблюдается тогда и только тогда, когда каждый путь, соединяющий любой узел в поддереве узла *w*, с любым узлом, не принадлежащим поддереву узла *w*, включает *v-w*. Другими словами, удаление *v-w* отделяет подграф, соответствующий поддереву узла w, от остальной части графа.

Для каждой вершины *v* мы используем рекурсивную функцию, вычисляющую минимальный номер в прямом порядке обхода, на который можно выйти через последовательность из нулевого или большего числа ребер дерева, за которыми следует одно обратное ребро из любого узла поддерева с корнем в вершине *v*. Если вычисленное число больше номера вершины v при прямом порядке обхода, то не существует ребра, связывающего потомок вершины *v* с ее предком, а это означает, что обнаружен мост.

Вычисления для каждой вершины достаточно просты: мы просматриваем списки смежных вершин, следя за тем, на какое минимальное число мы можем выйти, следуя по каждому ребру. Если обращение к рекурсивной функции для ребра *w-t* не приводит к обнаружению пути к узлу с меньшим номером прямого порядка обхода, чем аналогичный номер узла *t*, то ребро *w-t* является мостом.

Исследование алгоритма

Как указывалось ранее, асимптотическая временная сложность алгоритма линейная и составляет $O\,(V + E)$. Для исследования алгоритма создадим несколько графов с определенной суммой ребер и вершин, случайно соединенных этими ребрами между собой, и замерим время выполнения алгоритма Тарьяна. На рисунке 1 показан график зависимости времени выполнения программы от суммы ребер и вершин в графе.

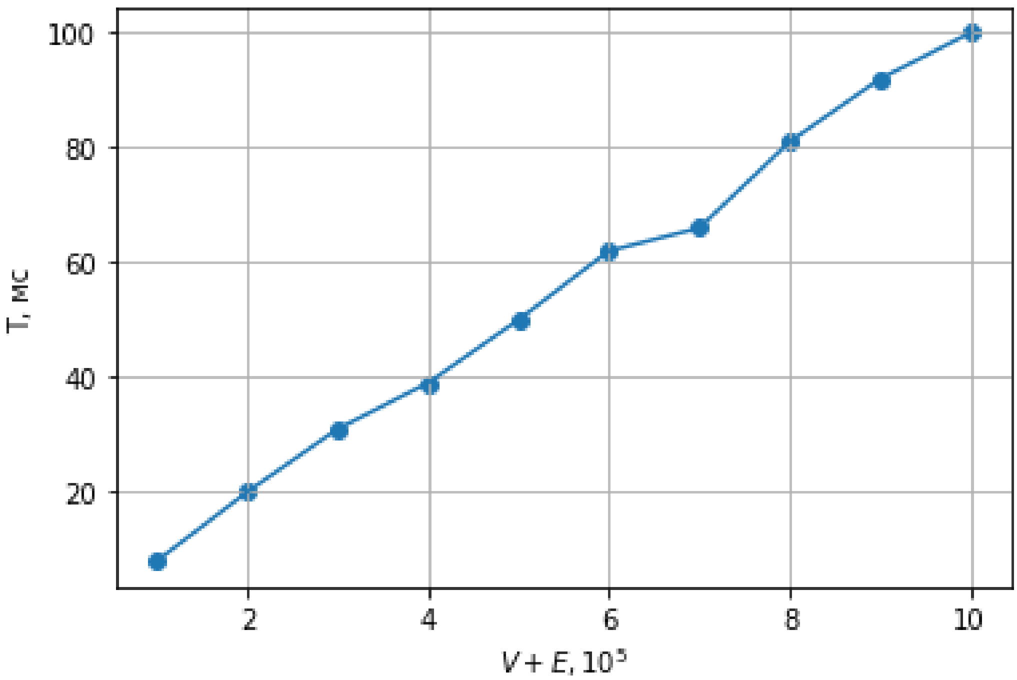

Рис. 1 – График зависимости времени от суммы ребер и вершин

Можно заметить, что график зависимости экспериментальной зависимости имеет линейный характер, что совпадает с теоретической зависимостью.

Заключение

В работе был рассмотрен алгоритм Тарьяна для поиска мостов в графе и его реализация на языке C++. Было обнаружено, что алгоритм имеет линейное асимптотическое время, а значит, он является эффективным для нахождения мостов в графе.

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.

Приложение 1

```
//graph.h

#ifndef TARJAN_S_BRIDGE_FINDING_ALGORITHM_GRAPH_H
#define TARJAN_S_BRIDGE_FINDING_ALGORITHM_GRAPH_H

#include <vector>

// A class that represents an undirected graph
class Graph{
private:
    int num_of_vertices;
    std::vector<std::list<int>> adj; // adjacency vector
private:
    int time;
public:
    Graph();
    explicit Graph(int num_of_vertices);
    explicit Graph(std::vector<std::list<int>> &adj);
    void add_edge(int v, int w);
    int get_number_of_edges();
    int get_number_of_vertices() const;
    bool edge_is_in_graph(int v, int w);
    void find_bridges();
private:
    void bridge_dfs(int u,
                    std::vector<bool> &visited,
                    std::vector<int> &disc,
                    std::vector<int> &low,
                    std::vector<int> &parent,
                    std::vector<std::pair<int, int>> &bridges);
};

Graph create_random_graph(int sum_of_edges_and_vertices);

#endif
```

```cpp
//graph.cpp

#include<iostream>
#include <list>
#include <vector>
#include <ctime>
#include <algorithm>
#include <cmath>
#include "graph.h"

Graph::Graph() {
    num_of_vertices = 0;
}

Graph::Graph(int num_of_vertices){
    this->num_of_vertices = num_of_vertices;
    adj = std::vector<std::list<int>>(num_of_vertices);
}

Graph::Graph(std::vector<std::list<int>> &adj){
    this->num_of_vertices = (int) adj.size();
    this->adj = adj;
}

void Graph::add_edge(int v, int w){
    adj[v].push_back(w);
    adj[w].push_back(v);
}

int Graph::get_number_of_edges(){
    int num = 0;
    for (auto & it : adj){
        num += it.size();
    }
    return num / 2; // each edge counts twice
}

int Graph::get_number_of_vertices() const{
    return num_of_vertices;
}

bool Graph::edge_is_in_graph(int v, int w){
    if (std::find(adj[v].begin(), adj[v].end(), w) != adj[v].end())
        return true;
    return false;
}
```

```cpp
// DFS based function to find all bridges. It uses recursive function bridge_dfs
void Graph::find_bridges()
{
    // Mark all the vertices as not visited
    std::vector<bool> visited(num_of_vertices, false);

    std::vector<int> disc(num_of_vertices);
    std::vector<int> low(num_of_vertices);
    std::vector<int> parent(num_of_vertices, -1);

    std::vector<std::pair<int, int>> bridges;
    time = 0;

    // Call the recursive helper function to find Bridges
    // in DFS tree rooted with vertex 'i'
    for (int i = 0; i < num_of_vertices; i++)
        if (!visited[i])
            bridge_dfs(i, visited, disc, low, parent, bridges);
}

// A recursive function that finds bridges using DFS traversal
void Graph::bridge_dfs(int u,
                       std::vector<bool> &visited,
                       std::vector<int> &disc,
                       std::vector<int> &low,
                       std::vector<int> &parent,
                       std::vector<std::pair<int, int>> &bridges) {

    // Mark the current node as visited
    visited[u] = true;

    // Initialize discovery time and low value
    disc[u] = low[u] = ++time;

    // Go through all vertices adjacent to this
    for (auto & v : adj[u]) {

        // If v is not visited yet, then recur for it
        if (!visited[v]) {
            parent[v] = u;
            bridge_dfs(v, visited, disc, low, parent, bridges);

            // Check if the subtree rooted with v has a
            // connection to one of the ancestors of u
            low[u] = std::min(low[u], low[v]);

            // If the lowest vertex reachable from subtree
            // under v is  below u in DFS tree, then u-v
            // is a find_bridges
            if (low[v] > disc[u]){
                bridges.emplace_back(u, v);
            }
        }
        // Update low value of u for parent function calls.
        else if (v != parent[u])
            low[u]  = std::min(low[u], disc[v]);
    }
}
```