Санкт-Петербургский Политехнический университет Петра Великого
Институт машиностроения, материалов и транспорта
Высшая школа автоматизации и робототехники

Курсовой проект

Дисциплина: Программирование на языках высокого уровня

Тема: Топологическая сортировка. Алгоритм Кана.

Разработал:
студент гр. 3331506/90401 Семенов Н.С.

Преподаватель Ананьевский М.С.

Санкт-Петербург
2022

Введение

В программировании часто возникает задача упорядочивания вершин бесконтурного орграфа – направленного графа, в котором отсутствуют направленные циклы, но могут быть параллельные пути, выходящие из одного узла и разными путями приходящие в конечный узел. Такие графы широко используются в компиляторах, машинном обучении, статистике и в других задачах, где необходимо корректно определить последовательность зависимых друг от друга действий. Пример такого графа представлен на рис. 1.

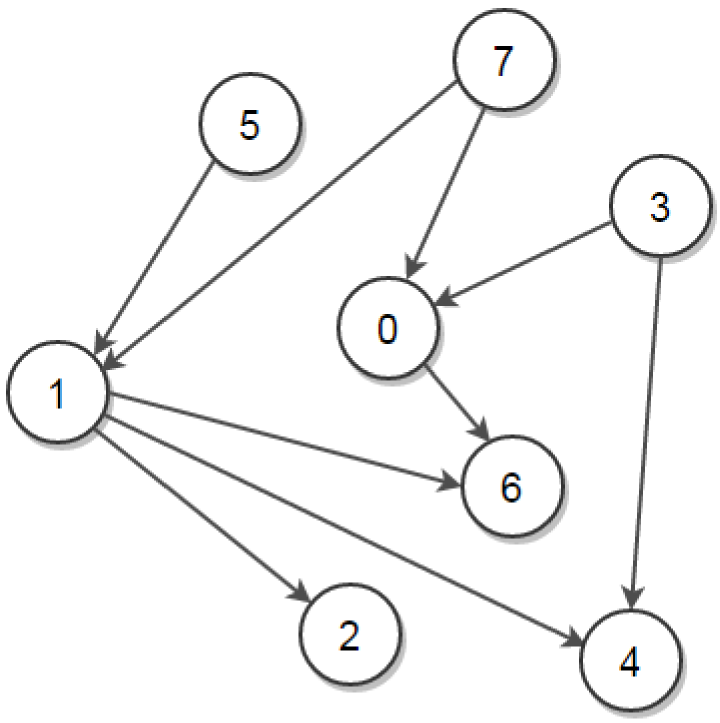


Рис. 1 – Бесконтурный ориентированный граф (не отсортирован)

Топологическая сортировка – это один из основных алгоритмов на графах, который применяется для решения множества более сложных задач. Топологическая сортировка применяется в самых разных ситуациях, например при распараллеливании алгоритмов, когда по некоторому описанию алгоритма нужно составить граф зависимостей его операций и, отсортировав его топологически, определить, какие из операций являются независимыми и могут выполняться параллельно (одновременно). Примером использования топологической сортировки может служить создание карты сайта, где имеет место древовидная система разделов.

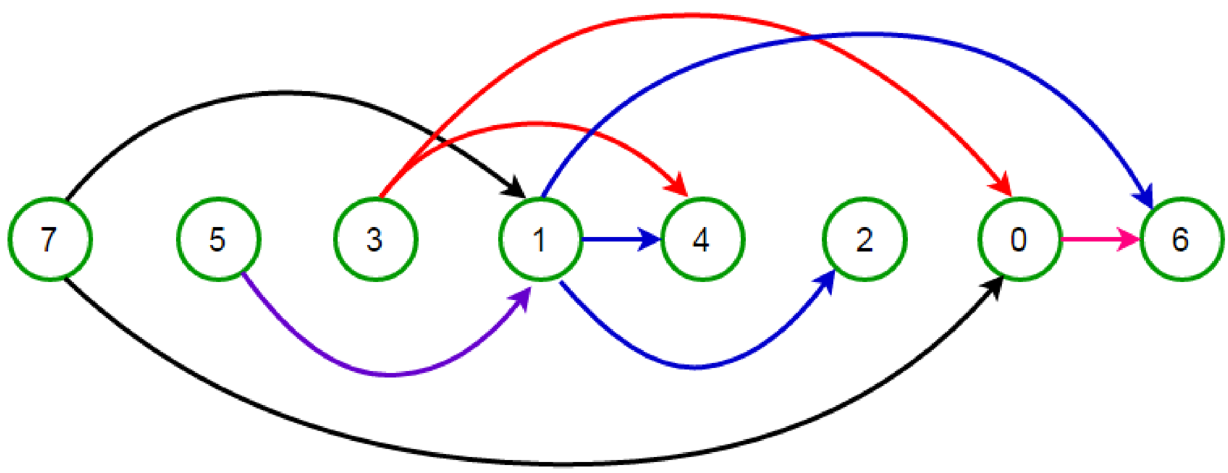
Задача топологической сортировки графа состоит в следующем: указать такой линейный порядок на его вершинах, чтобы любое ребро вело от вершины с меньшим номером к вершине с большим номером. Очевидно, что если в графе есть циклы, то такого порядка не существует.

Иными словами, линейное упорядочивание его вершин, такое, что для каждого направленного ребра ***BE*** от вершины ***B*** до вершины ***E*** ***B*** предшествует ***E*** в упорядочивании.

Для графа (рис. 1) существует несколько согласованных последовательностей его вершин, которые могут получены при помощи данного алгоритма сортировки. Результатом топологической сортировки данного графа могут быть следующие последовательности:

- 7 5 3 1 4 2 0 6
- 7 5 1 2 3 4 0 6
- 5 7 3 1 0 2 6 4
- И другие.

Обратим внимание, что для каждого направленного ребра ***BE*** ***B*** стоит перед ***E*** в порядке следования. Например, графическое представление топологического порядка [7, 5, 3, 1, 4, 2, 0, 6]:



Описание алгоритма

Один из вариантов решения поставленной задачи был впервые описан Артуром Каном в 1962. Алгоритм построен на выборе вершин в порядке, отражающем возможный вариант топологической сортировки.

Вначале необходимо найти перечень «начальных узлов» (у которых нет входящих рёбер) и поместить их в перечень ***Z***; хотя бы один такой узел должен существовать в непустом ациклическом графе. Псевдокод выглядит следующим образом:

F (сокр. от Final) -> Пустой список, который будет содержать отсортированные элементы;
Z (сокр. от Zero) -> Список всех вершин без входящих рёбер (т.е. имеющих степень вхождения 0);

```
пока список Z содержит вершины, выполнить {
    удалить вершину n из списка Z;
    добавить вершину n в конец списка F;
    для (каждой) вершины m с ребром p (от n до m) выполнить {
       удалить ребро p из графа;
       если вершина m не имеет других входящих рёбер, то {
              добавить вершину m в конец списка Z;
        }
     }
}
если граф имеет рёбра, то {
    вернуть ошибку «граф имеет хотя бы один цикл»
в ином случае {
    вернуть список F (топологически отсортированный граф)
}
```

Если граф является DAG (ориентированный ациклический граф), решение будет содержаться в списке ***F*** (решение не обязательно единственное). В противном случае граф имеет хотя бы один цикл и, следовательно, топологическая сортировка невозможна.

Отражая неединственность результирующей сортировки, перечень ***Z*** может быть простым набором, очередью или стеком. В зависимости от порядка удаления узлов ***n*** из перечня ***Z*** создаётся другое решение.

Реализация алгоритма

Алгоритм реализован на C++. Так как известно, что топологическая сортировка используется в том числе для более сложных алгоритмов на графах, то было принято решение реализовать алгоритм на C++, который позволяет создать интерфейс, который было бы удобно использовать не только для топологической сортировки. Созданы:

- Библиотека Graph, содержащая все основные функции и методы, такие как:
- конструкторы графа по заданному/случайно сгенерированному списку смежности, конструкторы копирования и перемещения;
- динамические массивы ***adjList*** (список сопряжения узлов, по сути хранит координаты всех рёбер, следовательно, весь граф) и ***degree*** (хранит степень вершины, т.е. число входящих в неё рёбер);
- функция ***doTopologicalSort***, являющаяся основой программы и отвечающая непосредственно за топологическую сортировку графа.

В функции ***main*** происходит ввод данных графа, вызов ранее обозначенной функции ***doTopologicalSort***, вывод конечного результата, а также подсчёт временных и пространственных затрат на выполнение алгоритма топологической сортировки. На выходе алгоритм выдает только лишь топологически отсортированную последовательность вершин, так как на практике в большинстве случаев этого достаточно – необходимо лишь знать правильную последовательность действий.

Анализ сложности алгоритма

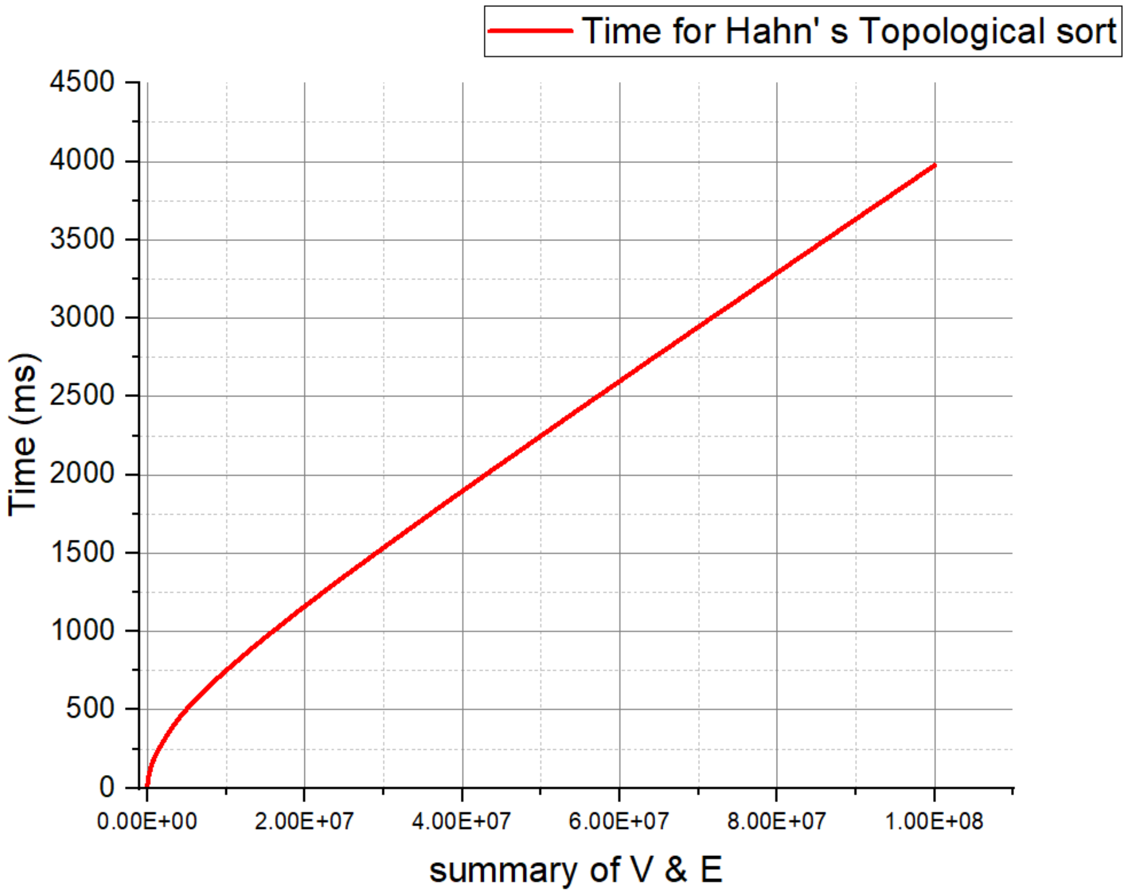
Теоретические затраты по времени

Сложность такого алгоритма соответствует сложности алгоритма поиска в глубину, то есть ***O(V+E)***, где V – число вершин, E – число рёбер. Это доказывается тем, что при проходе в глубину алгоритм лишь единожды проходит через текущую вершину, перемещая её затем в перечень ***F***, помечая её таким образом как посещённую.

В большинстве случаев количество рёбер в графе гораздо больше количества вершин. Поэтому определяющим фактором в сложности алгоритма является количество рёбер графа.

Анализ затрат по времени

На рисунке ниже представлена зависимость времени выполнения алгоритма топологической сортировки от (V+E) с входными данными в виде вектора векторов списка смежности.



Как видно из графика, сложность алгоритма топологической сортировки действительно соответствует сложности алгоритма поиска в глубину, то есть ***O(V+E)***.

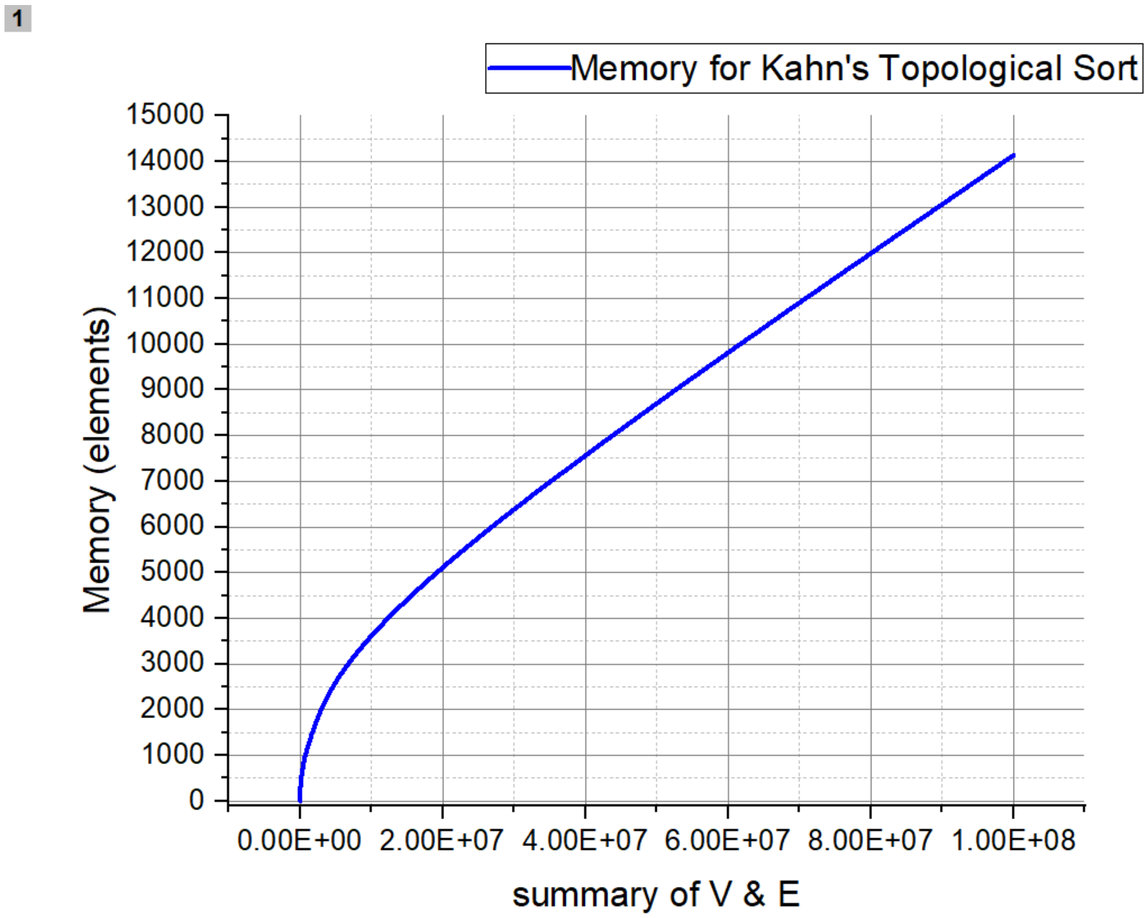
Теоретические затраты по памяти

Алгоритм топологической сортировки использует дополнительные массивы (в представленной работе – массивы ***F*** и ***Z***, отвечающие за формирование окончательного результата топологической сортировки и хранение вершин без входящих связей при каждой итерации соответственно). Размер двух массивов – количество элементов, не превышающее количество вершин. Следовательно, производительность по памяти ***O(V)***, где V – количество вершин.

Следует отметить, что в случае хранения графа в виде списка смежностей, мы достигаем рационального использования памяти, поскольку в строках списка смежностей хранятся лишь номера тех вершин, в которые ведут рёбра из текущей.

Анализ затрат по памяти

На рисунке ниже представлена зависимость затрат памяти алгоритма топологической сортировки от V с входными данными в виде вектора векторов списка смежности.



Как видно из графика, затраты по памяти алгоритма топологической сортировки соответствуют теоретическим, то есть ***O(V)***.

Заключение

В ходе выполнения курсового проекта была рассмотрена топологическая сортировка графа, реализация алгоритма которой была выполнена на языке C++.

Также был проведён анализ сложности алгоритма, затрат по времени и памяти на его выполнение, была рассмотрена область применения данной сортировки и ее особенности.

Список использованной литературы

1. Левитин А. В. Глава 5. Метод уменьшения размера задачи: Топологическая сортировка // Алгоритмы. Введение в разработку и анализ — М.: Вильямс, 2006. — С. 220–224. — 576 с. — ISBN 978-5-8459-0987-9
2. Кормен, Т., Лейзерсон, Ч., Ривест, Р., Штайн, К. Глава 22.4. Топологическая сортировка // Алгоритмы: построение и анализ = Introduction to Algorithms / Под ред. И. В. Красикова. — 2-е изд. — М.: Вильямс, 2005. — С. 632-635. — ISBN 5-8459-0857-4.
3. *Рафгарден Тим.* Совершенный алгоритм. Графовые алгоритмы и структуры данных. - СПб.: Питер, 2019. - 256 с.: ил. - (Серия «Библиотека программиста») — ISBN 978-5-4461-1272-2.
4. techiedelight.com/kahn-topological-sort-algorithm/

Приложение

Код библиотеки Graph:

```
#include "Graph.h"
#include <cstdlib>
#include <iostream>
#include <vector>
#include <cmath>
#include <algorithm>
#include <ctime>

Graph::Graph() {
    sumVertices = 0;
    sumEdges = 0;
}

Graph::Graph(int sumVertices) {
// Конструктор пустого листа смежностей
    this->sumVertices = sumVertices;
    adjList = std::vector<std::vector<int>>(sumVertices);
}

Graph::Graph(std::vector<std::vector<int>> &other_list) {
// Конструктор листа смежностей на основе
    sumVertices = other_list.size();
// исходного листа смежностей
    adjList = other_list;
    std::vector<int> temp(sumVertices, 0);
// Инициализируем степень вхождения вершины
    degree = temp;
    for (int row = 0; row < adjList.size(); row++) {
// Задаём степень вхождения каждой вершины
        for (int col = 0; col < adjList[row].size(); col++) {
// проходя по листу смежностей
            degree[adjList[row][col]]++;
        }
    }
}

Graph::Graph(const Graph& other_list) {
// Конструктор копирования
    sumVertices = other_list.sumVertices;
    adjList = other_list.adjList;
}

Graph::Graph(Graph&& other_list) {
// Конструктор перемещения
    sumVertices = other_list.sumVertices;
    adjList = other_list.adjList;
}

Graph::Graph(int sumEV, Type type) {
// Конструктор листа смежностей
    switch (type) {
// ацикличного ориентированного графа
        case RANDOM_DAG:
// на основе случайных данных
            srand(time(nullptr));
```

```
            sumVertices = (-1 + (int) ceil(sqrt(1 + 8.0 * sumEV))) / 2;
// Высчитываем количество вершин (плотный граф)
            sumEdges = sumEV - sumVertices;
// Количество рёбер графа
            adjList = std::vector<std::vector<int>>(sumVertices);
// Создаём лист смежностей нужного объёма
            int sumEdges_MAX_ON_VERTICE = sumEdges / sumVertices + 1;
// Максимальное количество исходящих рёбер для вершины
            int sumEdges_ADDED = 0;
// Всего рёбер добавлено
            int sumEdges_ADDED_ON_VERTICE = 0;
// Рёбер добавлено для одной вершины
            int ON_TARGET = 0;
// Инициализируем переменную для записи целевой вершины ребра
            for (int row = 0; row < sumVertices && sumEdges_ADDED < sumEdges;
row++) {      // Пробегаемся по каждой вершине пока рёбра не закончатся
                sumEdges_ADDED_ON_VERTICE = 0;
// Обнуляем счётчик рёбер для одной вершины
                std::vector<int> TARGET (sumVertices - row - 1);
// Создаём вектор возможных целей для рёбер текущей вершины
                ON_TARGET = 0;
// Обнуляем переменную для записи целевой вершины ребра
                for (int i = row + 1; i < sumVertices; i++){
// Генерируем вектор целей для текущей вершины
                    TARGET[i - row - 1] =  i;
// Т.к. всегда хотим получать DAG, то цели - из последующих вершин
                }
// Не забываем про смещение элементов на строке выше
                random_shuffle(TARGET.begin(), TARGET.end());
// Перемешиваем цели для случайной выборки далее
                for (int col=0; col < TARGET.size() &&
sumEdges_ADDED_ON_VERTICE       // Пробегаемся по текущей вершине, пока рёбра
или цели не закончатся
                        < sumEdges_MAX_ON_VERTICE; col++) {
                    ON_TARGET = TARGET[col];
// Выбираем случайную цель
                    adjList[row].push_back(ON_TARGET);
// Вносим в список смежностей
                    sumEdges_ADDED++;
// Увеличиваем счётчики рёбер
                    sumEdges_ADDED_ON_VERTICE++;
                }
                sort(adjList[row].begin(), adjList[row].end());
// Сортируем перечень целей рёбер текущей вершины в списке смежностей
            }
            std::vector<int> temp(sumVertices, 0);
// Инициализируем степень вхождения вершины
            degree = temp;
            for (int row = 0; row < adjList.size(); row++) {
// Задаём степень вхождения каждой вершины
                for (int col = 0; col < adjList[row].size(); col++) {
// проходя по листу смежностей
                    degree[adjList[row][col]]++;
                }
            }
    }
}

std::vector<int> Graph::doTopologicalSort(){
    std::vector<int> F;
// Вектор конечного результата
    std::vector<int> Z;
// Набор всех узлов без входящих ребер (degree = 0)
```

```
    int n = sumVertices;
// Получаем общее количество вершин графа
    for (int i = 0; i < n; i++) {
// Вершины без входящих рёбер добавляем в Z
        if (!degree[i]) {
            Z.push_back(i);

        }
    }
    while (!Z.empty()) {
// Пока перечень вершин без входящих рёбер не опустеет
        int n = Z.back();
// Удаляем узел `n` из `Z
        Z.pop_back();
        F.push_back(n);
// Добавляем `n` в конец `F`
        for (int m: adjList[n]){
            degree[m] -= 1;
// Удаляем из графа ребро `n` до `m
            if (!degree[m]) {
// Если `m` не имеет других входящих ребер, вставляем `m` в `Z
                Z.push_back(m);
            }
        }
    }
    for (int i = 0; i < n; i++) {
// Если в графе остались ребра, то в графе есть хотя бы один цикл
        if (degree[i]) {
            return {};
        }
    }
    return F;
}

int Graph::get_SumVertices() {
// Получение количества вершин графа
    return sumVertices;
}

int Graph::get_sumEdges() {
// Получение количества рёбер графа
    return sumEdges;
}

int Graph::get_adjList(){
// Получение размера списка смежностей
    int size=0;
    for (int row = 0; row < adjList.size(); row++) {
        for (int col = 0; col < adjList[row].size(); col++) {
            size++;
        }
    }
    return size;
}

void Graph::print_adjList() {
// Вывод списка смежностей на экран
    for (int row = 0; row < adjList.size(); row++) {
            std::cout << row << ": ";
            for (int col = 0; col < adjList[row].size(); col++) {
                std::cout << adjList[row][col] << " ";
            }
            std::cout << std::endl;
```

```
    }
}

void Graph::print_path(std::vector<int> path) {
// Вывод топологически отсортированного графа
    if (!path.empty()) {
        for (int i: path) {
            std::cout << i << " ";
        }
        std::cout << "" << std::endl;
    } else {
        std::cout << "Graph has at least one cycle. Topological sorting is
not possible" << std::endl;
    }
}
```

Код процедуры проверки времени выполнения алгоритма:

```
void doTopologicalSort_Measurement(int sumVE) {
// Процедура проверки времени выполнения топологической сортировки
    Graph g(sumVE, RANDOM_DAG);
    //g.print_adjList();
    int start = clock();
    g.print_path(g.doTopologicalSort());
    int end = clock();
    int time = (end - start) * 1000 / CLOCKS_PER_SEC;
    std::cout << "Summary of edges and vertices: " << g.get_SumVertices() +
g.get_sumEdges() << ", milliseconds: ";
    std::cout << time << std::endl;
    std::cout << "Size of adjacency list: " << g.get_adjList() << std::endl;
}
```