МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
"МИРЭА - Российский технологический университет"
РТУ МИРЭА

Институт искусственного интеллекта

Кафедра высшей математики

КУРСОВАЯ РАБОТА

по дисциплине
«Алгоритмы и теория сложности»

Тема курсовой работы

«Вычисление сильно связных компонентов (граф задан списками смежности)»

Студент группы КМБО-03-21 *Черников Т.Г.*

Руководитель курсовой работы *Драгилева И.П.*

Работа представлена к защите «27» 12 2023 г. *(подпись студента)*

«Допущен(ы) к защите» «27» 12 2023г. *(подпись руководителя)*

МОСКВА – 2023

МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
"МИРЭА - Российский технологический университет"
РТУ МИРЭА

Институт искусственного интеллекта

Кафедра высшей математики

Утверждаю
И.о. заведующего
кафедрой _____ Шатина А.В.

«_22_» декабря_2023г.

ЗАДАНИЕ
на выполнение курсовой работы
по дисциплине «Алгоритмы и теория сложности»

Студент Черников Т.Г. Группа *КМБО-03-21*

1. **Тема: «** Вычисление сильно связанных компонентов (граф задан списками смежности)».

2. Исходные данные: тестовые примеры и наборы данных для испытаний

3. Перечень вопросов, подлежащих обработке, и обязательного графического материала:
1) Алгоритм(ы) решения задачи на языке высокого уровня.
2) Получение и сравнение количественных результатов при различных исходных данных.
3) Асимптотические оценки временной сложности.

4. Срок представления к защите курсовой работы: до «23 » декабря 2023 г.

Задание на курсовую работу выдал « 1» октября 2023г . _____ (Драгилева)
Задание на курсовую работу получил «__1__» __10__ 2023г. _____ (Черников)

ОТЧЕТ
о курсовой работе
студента 3 курса учебной группы КМБО-03-21
института искусственного интеллекта
Российского технологического университета (МИРЭА)

Черникова Т.Г.

---

1. Задание на курсовую работу выполнил
   _____полностью_____ _____
   (указать: в полном объеме или частично)
2. Подробное содержание выполненных задач в ходе курсовой работы и достигнутые результаты: курсовая работа посвящена построению компонент сильной связности в ориентированном графе. Алгоритм реализован и протестирован. Доказана корректность алгоритма, приведена оценка временной сложности.
   Студент заслуживает оценки «отлично»

Руководитель курсовой работы

Старший преподаватель кафедры Высшей математики — _____ — Драгилева И.П..
(должность) (подпись) (фамилия и инициалы)

« 27 » 12 2023г.

Содержание

1. Введение

Тема вычисления компонентов сильной связности графа представляет собой одну из важных задач в области теории графов и компьютерных наук. Компоненты сильной связности являются фундаментальным понятием в теории орграфов, а их вычисление имеет широкий спектр практических применений, включая анализ сетей, оптимизацию маршрутов, анализ социальных и информационных сетей, а также в компьютерной биологии, биоинформатике и многих других областях.

В данной курсовой работе рассматривается метод вычисления компонентов сильной связности графа заданного списком смежности. Дается математическая оценка его асимптотической сложности и доказательство корректности, оценивается время выполнения алгоритма на практике.

2. Постановка задачи

1. Написать функцию, находящую в графе, заданном списком смежности, *все компоненты сильной связности*.

2. Доказать корректность используемого алгоритма нахождения компонентов сильной связности, произвести асимптотическую оценку его времени выполнения. Оценить время выполнения алгоритма на практике.

3. Проверить правильность выполнения функции для нахождения компонент сильной связности на тестовых наборах данных.

3. Теоретическая часть

3.1. Основные понятия

Компонентой сильной связности графа G называется такое подмножество V_1 его вершин, что для любых двух вершин u и v этого подмножества в графе G существует путь $u \rightsquigarrow v$ и не существует другого подмножества вершин $V_2 \supset V_1$, для которого выполняется это условие.

Транспонированный граф G^T - граф, полученный из G заменой всех его ребер (u, v) ребрами (v, u).

3.2. Описание используемых алгоритмов

Внутри основной программы для нахождения всех компонентов сильной связности используется подпроцедура, выполняющая вариацию алгоритма поиска в графе *Depth-first-search* [1, с. 43]. Данная вариация алгоритма последовательно просматривает все вершины графа и, в случае если эта вершина не отмечена посещенной, вызывает подпроцедуру *DFS-visit*, передавая ей на вход не отмеченную вершину. Подпроцедура *DFS-visit* отмечает переданную ей вершину v как посещенную и просматривает все смежные с ней вершины u. Если u не отмечена посещенной, подпроцедура вызывает саму себя, передавая на вход вершину u. После просмотра всех смежных вершин подпроцедура присваивает переданной ей на вход вершине число f, обозначающее количество вершин которым еще не присвоено это число.

Помимо описанной выше подпроцедуры, в поиске компонент сильной связности используется алгоритм транспонирования графа. Алгоритм состоит из последовательного просмотра всех смежных вершин v всех списков смежности вершин u. При нахождении какого-то ребра (u, v), в список смежности вершины v другого графа добавляется вершина u.

Нумерация вершин, осуществляемая подпроцедурой *DFS*, имеет важ-

```
f_count = |G.V| // глобальная переменная

DFS()
    for each vertix u ∈ G.V
        visited(u) = false

    // последовательный просмотр всех вершин графа
    for each vertix u ∈ G.V
        if visited(u) == false
            DFS-visit(G, u)

DFS-visit(G, u)
    visited(u) = true
    for each vertix v such as ∃ edge (u, v)
        if visited(v) == false
            DFS-visit(G, v)

    f(u) = f_count
    f_count = f_count - 1
```

Рис. 3.1. Псевдокод вариации алгоритма *DFS*, используемого в программе для нахождения всех компонент сильной связности графа

ное свойство (которое будет доказано в параграфе 3.3), позволяющее находить компоненты сильной связности: *для двух разных компонентов сильной связности* S_1 *и* S_2*, таких что для двух вершин* $u \in S_1$, $v \in S_1$ существует путь $u \rightsquigarrow v$, $\min\limits_{x \in S_1} f(x) < \min\limits_{x \in S_2} f(x)$. Это свойство и тот факт, что транспонированный граф имеет те же компоненты сильной связности что и исходный, позволяют получить алгоритм нахождения всех компонент сильной связности графа [2, с. 577], чей псевдокод представлен на рисунке 3.2. По итогу работы данного алгоритма определяется значение *SCC(v)* всех вершин v графа. Две вершины входят в одну компоненту сильной связности тогда и только тогда, когда их значения *SCC* совпадают.

```
1  SCC_count = 1 // глобальная переменная
2  SCC_search(G)
3      DFS(G) // нумерация всех вершин графа подпроцедурой на рисунке 3.1
4      G^T = transpose(G) // транспонирование графа G
5
6      // просмотр всех вершин в порядке возрастания значения f
7      for i = 1, 2, ..., |G^T.V|
8          u ∈ G^T.V such as f(u) = i
9          if visited(u) = false
10             DFS-visit'(G^T, u)
11             SCC_count = SCC_count + 1
12
13 // поиск в глубину
14 DFS-visit'(G^T, u)
15     visited(u) = true
16
17     for each vertix v such as ∃ edge (u, v)
18         if visited(v) = false
19             DFS-visit'(G^T, v)
20
21     SCC(u) = SCC_count
```

Рис. 3.2. Псевдокод алгоритма поиска всех компонент сильной связности в графе

3.3. Доказательство корректности

Доказательство того, что алгоритм на рисунке 3.1 сопоставляет каждой вершине число, следует из того, что цикл 8-10 для каждой не посещенной вершины графа вызывает функцию *DFS-visit*, которая присваивает переданной вершине число в строке 18.

Докажем что после выполнения алгоритма из рисунка 3.1 выполняется следующее утверждение [1, с. 60]:

Для любых двух разных компонентов сильной связности S_1, S_2, таких что для двух вершин $u \in S_1$, $v \in S_2$ существует ребро (u, v),

$$\min_{x \in S_1} f(x) < \min_{x \in S_2} f(x).$$

В ходе исполнения алгоритма *DFS* возможны два случая. Либо алгоритм первее отметит как посещенную какую-то вершину из S_1, либо какую-то вершину из S_2.

Допустим алгоритм *DFS* отметил некоторую вершину $s_1 \in S_1$ как посещенную раньше любой другой вершины из S_1 и раньше любой вершины $s_2 \in S_2$. Из-за того, что s_1 и u находятся в одной компоненте сильной связности, существует путь $s_1 \rightsquigarrow u$. Также по условию существует пусть $u \to v$ и, из-за того что v и s_2 находятся в одной компоненте сильной связности, существует путь $v \rightsquigarrow s_2$. Итого, существует путь $s_1 \rightsquigarrow s_2$. Это значит, что вызов *DFS-visit(s_1)* не закончится, пока не будет закончен вызов *DFS-visit(s_2)*. То есть вершине s_2 будет присвоен номер f раньше чем вершине s_1, следовательно он окажется больше номера f, присвоенного вершине s_1. Так как вершина s_1 была отмечена как посещенная раньше любой другой из S_1, ей присвоен номер f позже любой другой вершины из S_1, следовательно $\min\limits_{x \in S_1} f(x) = f(s_1) < f(s_2)$. Ввиду произвольности вершины $s_2 \in S_2$ $\min\limits_{x \in S_1} f(x) < \min\limits_{x \in S_2} f(x)$.

Допустим алгоритм *DFS* отметил некоторую вершину $s_2 \in S_2$ как посещенную раньше любой другой вершины из S_2 и раньше любой вершины $s_1 \in S_1$. Как показано в предыдущем параграфе, существует путь $s_1 \rightsquigarrow s_2$. Из-за этого путь $s_2 \rightsquigarrow s_1$ существовать не может так как это означало бы, что вершины из S_1 и S_2 образуют одну компоненту сильной связности, что противоречит условию. Так как путь $s_2 \rightsquigarrow s_1$ не существует, вызов *DFS-visit(s_2)* закончится раньше чем начнется вызов *DFS-visit(s_1)*. То есть вершине s_2 будет присвоен номер f раньше чем вершине s_1, следовательно его значение будет больше чем значение вершины s_1. Так как вершина s_2 была отмечена как посещенная раньше любой другой из S_2, ей присвоен номер f позже любой другой вершины из S_2, следовательно $f(s_1) < f(s_2) = \min\limits_{x \in S_2} f(x)$. Ввиду произвольности вершины $s_1 \in S_1$ $\min\limits_{x \in S_1} f(x) < \min\limits_{x \in S_2} f(x)$. Свойство доказано.

Важным прямым следствием данного свойства является тот факт, что

если с помощью алгоритма *DFS* отметить вершины графа G, то для двух любых компонентов сильной связности S_1 и S_2 графа G^T, таких что для двух вершин $u \in S_1$, $v \in S_2$ существует ребро (u, v),

$$\min_{x \in S_1} f(x) > \min_{x \in S_2} f(x).$$

Следствие выше позволяет доказать корректность работы алгоритма на рисунке 3.2, а именно, что по окончанию работы данного алгоритма каждой вершине будет присвоен такой номер *SCC*, что две вершины графа будут принадлежать одной компоненте сильной связности тогда и только тогда когда их значения *SCC* будут равны.

После нумерации вершин графа функцией *DFS* и транспонирования, алгоритм последовательно просматривает каждую вершину графа в порядке возрастания значения номера f у вершин. Если рассматриваемая вершина u не отмечена посещенной, то, ввиду порядка просмотра вершин, мы можем точно сказать, что все вершины с номером f меньше чем у u уже отмечены посещенными. Это значит, что рассматриваемая вершина имеет наименьшее значение номера f среди вершин, еще не отмеченных посещенными. Обозначим за S_1 компоненту связности, которой принадлежит вершина u. Допустим существует компонента сильной связности S_2, такая что ее вершины еще не были отмечены посещенными и существует вершина $v \in S_2$, такая что существует путь $u \rightsquigarrow v$. Из такого предположение, ввиду свойства указанного выше, следует, что $f(u) = \min_{x \in S_1} f(x) > \min_{x \in S_2} f(x)$, противоречие с утверждением о минимальности $f(u)$ среди вершин, еще не отмеченных посещенными. Это значит, что в ходе поиска в глубину мы не будем рассматривать вершины компонент сильной связности, отличной от S_1 и поэтому отметим текущем номером *SCC* только вершины компоненты сильной связности в которую входит вершина u.

3.4. Асимптотическая оценка времени работы алгоритма

Рассмотрим вариацию алгоритма *DFS* на рисунке 3.1. Алгоритм проходит по всем вершинам не более одного раза и для каждой вершины u просматривает каждое ребро вида (u, v). Всего таких ребер не более $O(E)$. Итого, время работы алгоритма оценивается в $O(V + E)$

При транспонировании графа алгоритм просматривает все $O(V)$ списков. В каждом списке столько элементов, сколько исходящих из соответствующей вершины ребер. Всего таких ребер $O(E)$. Итого, транспонированный граф вычисляется за $O(V + E)$.

В алгоритме на рисунке 3.2 нахождения всех компонент сильной связности после нумерации вершин и транспонирования графа выполняется вариация *DFS*, но с определенным порядком рассмотрения вершин. Время работы такое же как у алгоритма на рисунке 3.1 - $O(V + E)$.

Итого, алгоритм поиска всех компонент сильной связности выполняется за время $O(V + E)$.

4. Практическая часть

4.1. Тестирование программы на малых данных

Практическая реализация алгоритма на рисунке 3.2 поиска всех компонент сильной связности представляет собой функцию, принимающую на вход ссылку на вектор *SCC* в i-ой ячейке которого после окончания программы будет записан номер компоненты связности к которой принадлежит вершина под номером i. Также функция принимает на вход граф в виде вектора списков. После исполнения функции представим вектор *SCC* в более удобном формате как массив списков в котором перечислены вершины состоящие в одной компоненте связности. Результаты работы функции для графов полученных из тестовых файлов малых размеров представлены на рисунке

```
test 1:
SCC as function: 2 0 1 2 0 1 2 0 1

SCC as set:
1: 8 5 2
2: 9 6 3
3: 7 4 1

test 2:
SCC as function: 0 0 0 2 2 1 1 1

SCC as set:
1: 3 2 1
2: 8 7 6
3: 5 4

test 3:
SCC as function: 2 2 2 3 1 0 0 0

SCC as set:
1: 8 7 6
2: 5
3: 3 2 1
4: 4

test 4:
SCC as function: 1 1 1 1 0 1 1 1

SCC as set:
1: 5
2: 8 7 6 4 3 2 1

test 5:
SCC as function: 0 1 2 1 1 2 3 3 3 3 3 3

SCC as set:
1: 1
2: 5 4 2
3: 6 3
4: 12 11 10 9 8 7
```

Рис. 3.3. Результаты тестов функции на графах полученных из тестовых файлов малых размеров

3.3.

4.2. Тестирование программы на больших данных

На рисунке 4.1 представлены временные результаты работы функции в миллисекундах для графа с большим количеством вершин и ребер. Функция запускалась на одном и том же графе несколько раз для выведения средней продолжительности исполнения.

```
experiment 0: 3471.05
experiment 1: 4059.89
experiment 2: 3900.98
experiment 3: 3988.89
experiment 4: 3878.34
avg of experiments: 3859.83
```

Рис. 4.1. Временные результаты в миллисекундах теста функции нахождения всех компонент сильной связности на большом графе

5. Заключение

В рамках курсовой работы был рассмотрен метод нахождения всех компонент сильной связности в графе представленном в виде списков смежности. Получены доказательства корректности и асимптотическая оценка времени работы программы. Программа была протестирована на малых и больших по объему данных.

Список литературы

1. *Roughgarden, T.* Algorithms Illuminated: Graph algorithms and data structures. Part 2 / T. Roughgarden. Algorithms Illuminated — Soundlikeyourself Publishing LLC — 2018.

2. Introduction to Algorithms, fourth edition / T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein — MIT Press — 2022.

Приложения

```
//         SCC-finder.h

#include <iostream>
#include <vector>
#include <queue>
#include <stack>
#include <forward_list>

void make_adjacency_list(std::vector<std::forward_list<int>> &graph,
↪  std::istream &istream) {
        int from; istream >> from;
        --from;
        int to; istream >> to;
        --to;
        do {
                if (graph.size() ⩽ from) {
                        graph.resize(from + 1);
                }
                graph[from].push_front(to);

                istream >> from;
                --from;
                istream >> to;
                --to;
        } while(!istream.eof());

        graph.shrink_to_fit();
```

```
}

void finish_time(std::forward_list<int> &finish_time_q, const
↪  std::vector<std::forward_list<int>> &graph) {
        auto q_last_element = finish_time_q.before_begin();
        std::stack<std::pair<int, std::forward_list<int>::const_iterator>>
        ↪  stack;
        std::vector<bool> visited = std::vector<bool>(graph.size());

        for (int i = 0; i < graph.size(); ++i) {
                if (!visited[i]) {
                        stack.push(std::make_pair(i, graph[i].begin()));

                        while(!stack.empty()) {
                                int current = stack.top().first;
                                visited[current] = true;
                                bool all_descendants_visited = true;
                                for (auto iterator = stack.top().second;
                                ↪  iterator != graph[current].end();
                                ↪  ++iterator) {
                                        if (!visited[*iterator]) {
                                                ++stack.top().second;
                                                stack.push(std::make_pair(*iterator
                                                ↪  graph[*iterator].begin()));
                                                all_descendants_visited =
                                                ↪  false;
                                                break;
                                        }
```

```
                                }

                                if (all_descendants_visited) {
                                        stack.pop();
                                        finish_time_q.push_front(current);
                                }
                        }
                }
        }
}

std::vector<std::forward_list<int>> graph_transposition(const
↪  std::vector<std::forward_list<int>> &graph) {
        std::vector<std::forward_list<int>> transposed_graph =
        ↪  std::vector<std::forward_list<int>>(graph.size());
        for (int i = 0; i < graph.size(); ++i) {
                for (auto iterator = graph[i].begin(); iterator ≠
                ↪  graph[i].end(); ++iterator) {
                        transposed_graph[*iterator].push_front(i);
                }
        }
        return transposed_graph;
}

void find_SCC(std::vector<int> &SCC, std::vector<std::forward_list<int>>
↪  &graph) {
        std::forward_list<int> finish_time_q;
        finish_time(finish_time_q, graph);
```

```
graph = graph_transposition(graph);


SCC = std::vector<int>(graph.size(), -1);
int SCC_number = 0;
std::stack<std::pair<int, std::forward_list<int>::const_iterator>>
↪  stack;
std::vector<bool> visited = std::vector<bool>(graph.size());
for (auto iterator1 = finish_time_q.begin(); iterator1 ≠
↪  finish_time_q.end(); ++iterator1) {
        if (SCC[*iterator1] == -1) {
                stack.push(std::make_pair(*iterator1,
                ↪  graph[*iterator1].begin()));


                while(!stack.empty()) {
                        int current = stack.top().first;
                        visited[current] = true;
                        SCC[current] = SCC_number;
                        bool all_descendants_visited = true;
                        for (auto iterator2 = stack.top().second;
                        ↪  iterator2 ≠ graph[current].end();
                        ↪  ++iterator2) {
                                if (!visited[*iterator2]) {
                                        ++stack.top().second;
                                        stack.push(std::make_pair(*iterator
                                        ↪  graph[*iterator2].begin()));
                                        all_descendants_visited =
                                        ↪  false;
                                        break;
```

```
                                        }
                                }

                                if (all_descendants_visited) { stack.pop();
                                ↪  }
                        }
                        ++SCC_number;
                }
        }
}

std::vector<std::forward_list<int>> SCC_function_to_sets(const
↪  std::vector<int> &SCC) {
        std::vector<std::forward_list<int>> sets;
        for (int i = 0; i < SCC.size(); ++i) {
                if (sets.size() ⩽ SCC[i]) {
                        sets.resize(SCC[i] + 1);
                }
                sets[SCC[i]].push_front(i);
        }
        return sets;
}

// CMakeLists.txt

cmake_minimum_required(VERSION 3.0.0)
project(StronglyConnectedComponents VERSION 0.1.0 LANGUAGES C CXX)
```

```
include(CTest)
enable_testing()

add_executable(MakeAdjacencyListTest tests/make-adjacency-list-test.cpp)
add_executable(FinishTimeTest tests/finish-time-test.cpp)
add_executable(GraphTranspositionTest tests/graph-transposition-test.cpp)
add_executable(FindSCCTest tests/find-SCC-test.cpp)
add_executable(LargeGraphSCCTest tests/large-graph-SCC-test.cpp)

set(CPACK_PROJECT_NAME ${PROJECT_NAME})
set(CPACK_PROJECT_VERSION ${PROJECT_VERSION})
include(CPack)
```

```
// tests/object-string.h

#include <vector>
#include <string>
#include <sstream>
#include <forward_list>

std::string graph_string(const std::vector<std::forward_list<int>> &graph) {
        std::stringstream ss;

        if (graph.size() != 0) {
                auto iterator = graph[0].begin();
                ss << "1: ";
                if (iterator != graph[0].end()) {
                        ss << *iterator + 1;
```

```
                ++iterator;
            }
            for (; iterator != graph[0].end(); ++iterator) {
                ss << " " << *iterator + 1;
            }
        }
        for (int i = 1; i < graph.size(); ++i) {
            ss << "\n" << i + 1 << ": ";
            auto iterator = graph[i].begin();
            if (iterator != graph[i].end()) {
                ss << *iterator + 1;
                ++iterator;
            }
            for (; iterator != graph[i].end(); ++iterator) {
                ss << " " << *iterator + 1;
            }
        }
        return ss.str();
}

std::string vertices_string(const std::forward_list<int> &vertices) {
        std::stringstream ss;

        auto iterator = vertices.begin();
        if (iterator != vertices.end()) {
            ss << *iterator + 1;
            ++iterator;
        }
```

```
        for (; iterator != vertices.end(); ++iterator) {
                ss << " " << *iterator + 1;
        }
        return ss.str();
}

std::string vector_string(const std::vector<int> &vector) {
        std::stringstream ss;

        if (vector.size() != 0) {
                ss << vector[0];
        }
        for (int i = 1; i < vector.size(); ++i) {
                ss << " " << vector[i];
        }
        return ss.str();
}

// tests/make-adjacency-list-test.cpp

#include <iostream>
#include <fstream>
#include <sstream>
#include "../SCC-finder.h"
#include "object-string.h"

int main() {
        for (int i = 1; i <= 5; ++i) {
```

```
            std::stringstream path;
            path << "../tests/test-matrices/problem8.10test" << i <<
            ↪  ".txt";
            std::ifstream file;
            file.open(path.str());
            std::vector<std::forward_list<int>> graph;
            make_adjacency_list(graph, file);
            file.close();
            std::cout << "test " << i << ":\n" << graph_string(graph) <<
            ↪  "\n\n";
        }
}

// tests/large-graph-SCC-test.cpp

#include <iostream>
#include <forward_list>
#include <fstream>
#include <chrono>
#include "../SCC-finder.h"
#include "../tests/object-string.h"

int main() {
        std::vector<std::forward_list<int>> graph;
        std::ifstream file;
        file.open("../tests/test-matrices/problem8.10.txt");
        make_adjacency_list(graph, file);
        file.close();
```

```
        std::vector<int> SCC;

        int number_experements = 5;
        std::vector<double> durations =
        ↪  std::vector<double>(number_experements);
        for (int i = 0; i < number_experements; ++i) {
                auto start = std::chrono::high_resolution_clock::now();
                find_SCC(SCC, graph);
                auto stop = std::chrono::high_resolution_clock::now();

                std::chrono::duration<double, std::milli> ms_double = stop -
                ↪  start;
                durations[i] = ms_double.count();
                std::cout << "experiment " << i << ": " << durations[i] <<
                ↪  "\n";
        }

        double avg = 0;
        for (int i = 0; i < number_experements; ++i) {
                avg += durations[i] / number_experements;
        }
        std::cout << "avg of experiments: " << avg << "\n";
}

// tests/graph-transposition-test.cpp

#include <iostream>
```

```cpp
#include <sstream>
#include <fstream>
#include "../SCC-finder.h"
#include "object-string.h"

int main() {
        for (int i = 1; i <= 5; ++i) {
                std::stringstream path;
                path << "../tests/test-matrices/problem8.10test" << i <<
                ↪  ".txt";
                std::ifstream file;
                file.open(path.str());
                std::vector<std::forward_list<int>> graph;
                make_adjacency_list(graph, file);
                file.close();

                graph = graph_transposition(graph);
                std::cout << "test " << i << ":\n" << graph_string(graph) <<
                ↪  "\n\n";
        }
}

// tests/finish-time-test.cpp

#include <iostream>
#include <vector>
#include <fstream>
#include <sstream>
```

```
#include <forward_list>
#include "../SCC-finder.h"
#include "object-string.h"

int main() {
        for (int i = 1; i <= 5; ++i) {
                std::stringstream path;
                path << "../tests/test-matrices/problem8.10test" << i <<
                ↪  ".txt";
                std::ifstream file;
                file.open(path.str());
                std::vector<std::forward_list<int>> graph;
                make_adjacency_list(graph, file);
                file.close();

                std::forward_list<int> finish_time_q;
                finish_time(finish_time_q, graph);
                std::cout << "test " << i << ":\n" <<
                ↪  vertices_string(finish_time_q) << "\n\n";
        }
}

// tests/find-SCC-test.cpp

#include <iostream>
#include <sstream>
#include <fstream>
#include "../SCC-finder.h"
```

```
#include "object-string.h"

int main() {
        for (int i = 1; i ⩽ 5; ++i) {
                std::stringstream path;
                path << "../tests/test-matrices/problem8.10test" << i <<
                ↪  ".txt";
                std::ifstream file;
                file.open(path.str());
                std::vector<std::forward_list<int>> graph;
                make_adjacency_list(graph, file);
                file.close();

                std::vector<int> SCC;
                find_SCC(SCC, graph);
                std::cout << "test " << i << ":\n" << "SCC as function: " <<
                ↪  vector_string(SCC) << "\n\n" << "SCC as set:" << "\n" <<
                ↪  graph_string(SCC_function_to_sets(SCC)) << "\n\n";
        }
}
```