

Задача 4

Файл: graph.hpp

```
#ifndef graph_hpp
#define graph_hpp

#include <iostream>
#include <list>
#include <stack>
#include <vector>
#include "pair.hpp"

class Graph
{
private:
    int v_;
    std::vector<std::list<int>>> adj;

public:
    Graph(int v);
    std::list<int> DFS(int v, std::vector<bool> &visited, std::list<int> &list, std::vector<int>
        &numSCC, int count);
    void addEdge(int v, int adjV);
    void fillOrder(int v, std::vector<bool> &visited, std::stack<int> &Stack);
    void printGraph();
    void printCss(SCCPair &pair);
    Graph appealGraph();
    Graph metaGraph();
    SCCPair SCC();
};

#endif /* graph_hpp */
```

Файл: graph.cpp

```
#include "graph.hpp"

Graph::Graph(int v)
{
    this->v_ = v;
    adj.resize(v_);
}

std::list<int> Graph::DFS(int v, std::vector<bool> &visited, std::list<int> &list,
    std::vector<int> &numSCC, int count)
{
    visited[v] = true;
    numSCC[v] = count;
    list.push_back(v);

    for (auto i = adj[v].begin(); i != adj[v].end(); ++i)
    {
        if (!visited[*i])
        {
            DFS(*i, visited, list, numSCC, count);
        }
    }

    return list;
}

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

Graph Graph::appealGraph()
{
    Graph appealG(v_);

    for (int v = 0; v < v_; v++)
    {
        for (auto i = adj[v].begin(); i != adj[v].end(); ++i)
```

```

        {
            appealG.adj[*i].push_back(v);
        }
    }

    return appealG;
}

void Graph::printGraph()
{
    for (int v = 0; v < v_; v++)
    {
        std::cout << v << " ";
        for (auto i = adj[v].begin(); i != adj[v].end(); ++i)
        {
            std::cout << *i << " ";
        }
        std::cout << '\n';
    }
}

void Graph::printCss(SCCPair &pair)
{
    for (int v = 0; v < pair.sccList.size(); v++)
    {
        std::cout << "SCC " << v << ": ";
        for (auto i = pair.sccList[v].begin(); i != pair.sccList[v].end(); ++i)
        {
            std::cout << *i << " ";
        }
        std::cout << '\n';
    }
}

void Graph::fillOrder(int v, std::vector<bool> &visited, std::stack<int> &Stack)
{
    visited[v] = true;

    for (auto i = adj[v].begin(); i != adj[v].end(); i++)
    {
        if (!visited[*i])
        {
            fillOrder(*i, visited, Stack);
        }
    }

    Stack.push(v);
}

SCCPair Graph::SCC()
{
    std::stack<int> stack;

    std::vector<bool> visited(v_);

    for (int i = 0; i < v_; i++)
    {
        visited[i] = false;
    }

    for (int i = 0; i < v_; i++)
    {
        if (visited[i] == false)
        {
            fillOrder(i, visited, stack);
        }
    }

    Graph gr = appealGraph();

    for (int i = 0; i < v_; i++)
    {
        visited[i] = false;
    }

    int count = 0;

```

```

std::vector<int> numSCC(v_);
std::vector<std::list<int>> listsOfSCC;

while (stack.empty() == false)
{
    int v = stack.top();
    stack.pop();

    if (visited[v] == false)
    {
        std::list<int> listCss;
        gr.DFS(v, visited, listCss, numSCC, count);
        listsOfSCC.push_back(listCss);
        count++;
    }
}

return SCCPair(listsOfSCC, numSCC);
}

Graph Graph::metaGraph()
{
    SCCPair graphScc = SCC();

    Graph metaGraph(graphScc.sccList.size());

    std::vector<bool> used(graphScc.sccList.size(), false);

    for (int n = 0; n < graphScc.sccList.size(); n++)
    {
        for (auto v = graphScc.sccList[n].begin(); v != graphScc.sccList[n].end(); v++)
        {
            for (auto i = adj[*v].begin(); i != adj[*v].end(); i++)
            {
                if (n != graphScc.numSCC[*i] && !used[graphScc.numSCC[*i]])
                {
                    metaGraph.addEdge(n, graphScc.numSCC[*i]);
                    used[graphScc.numSCC[*i]] = true;
                }
            }
        }

        for (auto iter = metaGraph.adj[n].begin(); iter != metaGraph.adj[n].end(); iter++)
        {
            used[*iter] = false;
        }
    }

    return metaGraph;
}

```

Файл: sccpair.hpp

```

#ifndef pair_hpp
#define pair_hpp

#include <list>
#include <vector>

struct SCCPair
{
    std::vector<std::list<int>> sccList;
    std::vector<int> numSCC;

    SCCPair(std::vector<std::list<int>> sccList, std::vector<int> sccCount)
    {
        this->sccList = sccList;
        this->numSCC = sccCount;
    }
};

#endif /* pair_hpp */

```

Файл: main.cpp

```
#include <iostream>
#include <sstream>
#include "graph.hpp"
#include "pair.hpp"

using namespace std;

int main()
{
    int size = 0;

    cin >> size;
    cin.ignore();
    Graph graph(size);

    for (int i = 0; i < size; i++)
    {
        string str;
        getline(cin, str);
        std::stringstream stream(str);

        std::string v;
        stream >> v;
        std::string u;

        int v1 = stoi(v);

        while (u != "-1")
        {
            stream >> u;
            if (u != "-1")
            {
                int u1 = stoi(u);
                graph.addEdge(v1, u1);
            }
        }
        cout << "\n";
        SCCPair pair = graph.SCC();
        graph.printCss(pair);

        std::cout << "\nMeta-graph:\n";
        graph.metaGraph().printGraph();
        return 0;
    }
}
```

АЛГОРИТМ ОБРАЩЕНИЯ ГРАФА

1) Псевдокод:

```
Ф-я reverseGraph ( graph ) { //Вход: 'graph' - список смежности
// представленный в виде массива списков list[0...n] ('v' кол-во
// вершин в графе), где 's' список хранит вершины смежные
// с вершиной 's'; Выход: список смежности обращения графа
// 'graph'.
```

```
    reverseGraph := list[0...n];
```

```
    для каждой вершины v в graph:
```

```
        для каждого э-та vertex списка смежности adjList
        вершины v:
```

```
            reverseGraph.adjList[vertex].push-back(v);
```

```
    вернуть reverseGraph;
```

```
}
```

2) Обоснование корректности

Для обращения графа необходимо поменять направление связи между смежными вершинами.

Пример:

Дан ориграф $A \rightarrow B$, тогда обрат. ориграф: $A \leftarrow B$

Я реализовала алгоритм в котором происходит обход списка смежности данного графа, в ходе которого в новый список смежности попадает вершина под индексом смежной вершины, таким образом меняется направление связи.

Р-м результат работы алгоритма на примере графа, представленного в #3 и 2.

Input	Визуализация
<pre>6 0 1 -1 1 2 3 -1 2 0 3 -1 3 4 -1 4 3 -1 5 5 -1</pre>	
Output	
<pre>0 2 1 0 2 1 3 1 2 4 4 3 5 5</pre>	

=> Алгоритм корректен

3) Оценка времени работы

$\exists T(V, E)$ - время работы ф-и `арреа/Graph`

- Проход по списку смежности выполняется за $T_E(V, E) = O(|V| + |E|)$

- Функция `push_back` за $T_{pb} = O(1)$

Следовательно, получим $T(V, E) = O(|V| + |E|)$ - время работы алгоритма `арреа/Graph`

Алгоритм построения метаграфа

1) Псевдокод:

ф-я `metagraph (graph, sccList, sccNum)` {

// Вход: `graph` (описан в псевдокоде алгоритма обращения графа);

// `sccList` - массив списков, где индекс - номер ССК,

// а значение под этим индексом - список вершин

// принадлежащих этому ССК; `numSCC[0...v-1]` - массив

// целых чисел, где индекс - вершина, а значение номер

// ССК которой принадлежит эта вершина. Вход: метаграф

// графа `graph`.

`metaGraph := list[0... sccList.size];`

`used := bool[sccCount];`

для каждой n ССК графа `graph`:

 для каждой вершины v в `sccList[n]`:

 для каждого смежного эл-та i вершины v :

 если $n \neq \text{numSCC}[i]$ && $\neg \text{used}[\text{numSCC}[i]]$:

`used[numSCC[i]] = true;`

 добавить в `metaGraph` вершину n со смежной `numSCC[i]`;

 для каждого смежного эл-та `elem` в `metaGraph[n]`:

`used[elem] = false;`

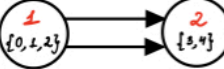
вернуть `metaGraph`;

2) Обоснование корректности

Для построения метаграфа необходимо восстановить связь между ССК с помощью связей вершин входящих в них.

Для этого разработанный мной алгоритм обходит каждый список ССК графа, затем проходит по всем ребрам исходящим из всех этих вершин (по счетным) входящих в ССК.

Если ребро соединяет разные номера компонент метаграфа, то проводим ребро. Также во избежание одинаковых записей

в итоговом списке смежности (избавляемся от кратных ребер, например: ) я завел массив типа bool,

при помощи которого буду помечать ребра которые уже были проведены, т.е. присваивать значение true. Перед переходом к обработке следующей ССК мы присваиваем значения false тем элементам, которые ранее отметили true.

Таким образом, в итоге мы получим метаграф =>

=> алгоритм корректен.

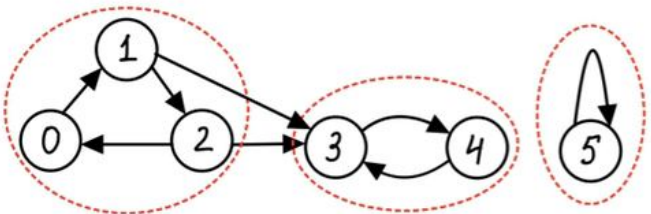

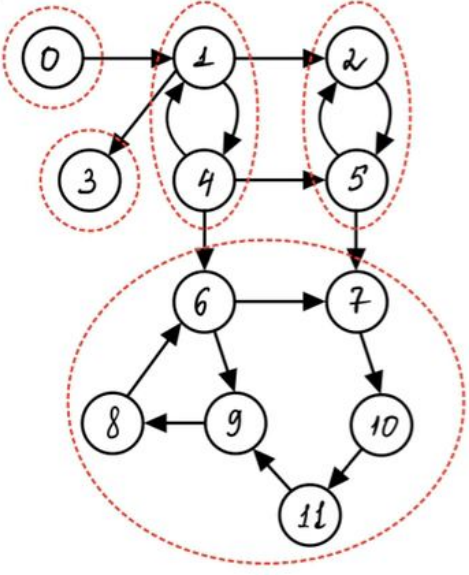
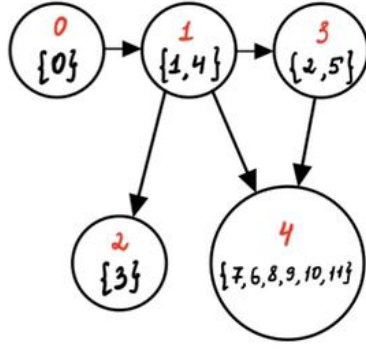
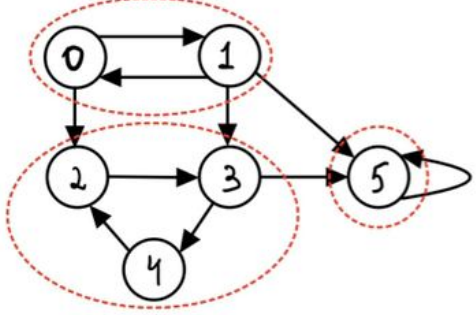
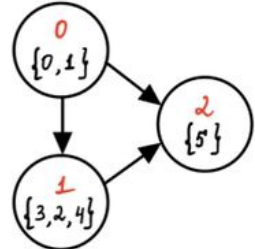
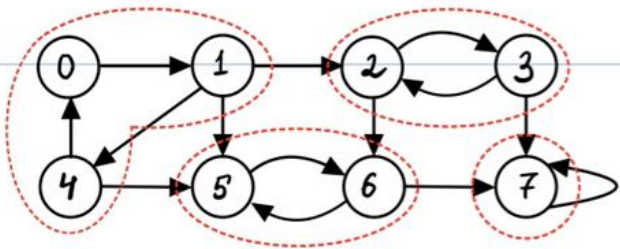
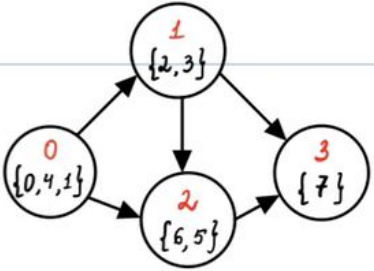
3) Оценка времени работы

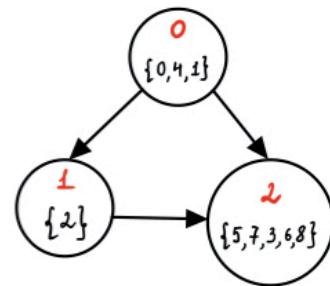
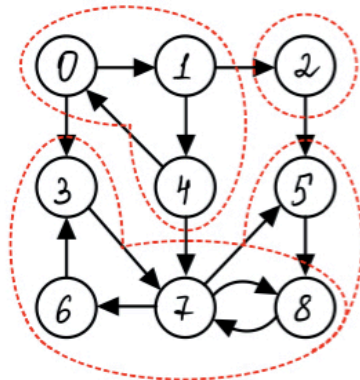
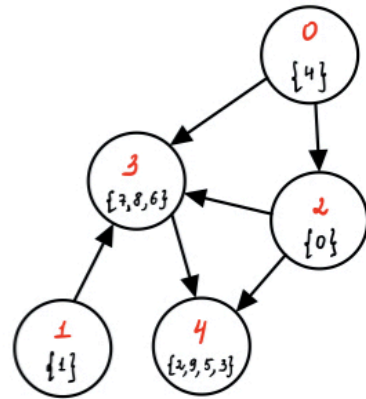
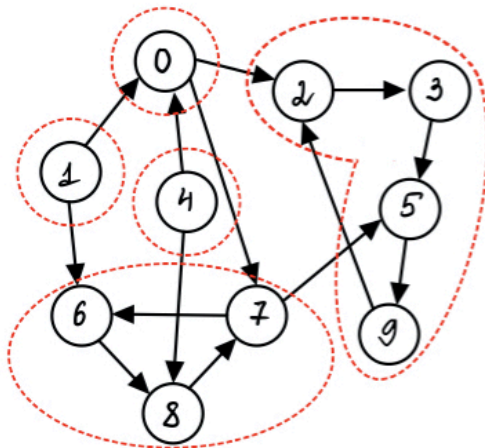
$T(V, E)$ - время работ ф-и `metaGraph`

- Проход по списку смежности выполняется за $T_c(V, E) = O(|V| + |E|)$
- Ф-я `addEdge()` выполняется за константное время - $O(1)$
- Операции присваивание и обращение по индексу также выполняются за константное время - $O(1)$

Следовательно, получим $T(V, E) = O(|V| + |E|)$ - время работы алгоритма `metaGraph`.

Визуализация построения метографа

Исходный граф	Метограф
<div data-bbox="244 302 359 331">из ДЗ 2</div> 	
<div data-bbox="244 573 359 602">ДПВ стр. 93</div> 	
<div data-bbox="244 1209 359 1238">ЛЕКЦИЯ</div> 	
<div data-bbox="244 1603 359 1632">Кормен 653</div> 	



Тесты

Input	Output
6 0 1 -1 1 2 3 -1 2 0 3 -1 3 4 -1 4 3 -1 5 5 -1	SCC 0: 5 SCC 1: 0 2 1 SCC 2: 3 4 Meta-graph: 0 1 2 2
12 0 1 -1 1 2 3 4 -1 2 5 -1 3 -1 4 1 5 6 -1 5 2 7 -1 6 7 9 -1 7 10 -1 8 6 -1 9 8 -1 10 11 -1 11 9 -1	SCC 0: 0 SCC 1: 1 4 SCC 2: 3 SCC 3: 2 5 SCC 4: 7 6 8 9 11 10 Meta-graph: 0 1 1 3 2 4 2 3 4 4
6 0 1 2 -1 1 0 3 5 -1 2 3 -1 3 4 5 -1 4 2 -1 5 5 -1	SCC 0: 0 1 SCC 1: 3 2 4 SCC 2: 5 Meta-graph: 0 1 2 1 2 2
8 0 1 -1 1 2 4 5 -1 2 3 6 -1 3 2 7 -1 4 0 5 -1 5 6 -1 6 5 7 -1 7 7 -1	SCC 0: 0 4 1 SCC 1: 2 3 SCC 2: 6 5 SCC 3: 7 Meta-graph: 0 1 2 1 2 3 2 3 3
10 0 2 7 -1 1 6 -1 2 3 -1 3 5 -1 4 0 8 -1 5 9 -1 6 8 -1 7 5 6 -1 8 7 -1 9 2 -1	SCC 0: 4 SCC 1: 1 SCC 2: 0 SCC 3: 7 8 6 SCC 4: 2 9 5 3 Meta-graph: 0 2 3 1 3 2 4 3 3 4 4
9 0 1 3 -1 1 2 4 -1 2 5 -1 3 7 -1 4 7 0 -1 5 8 -1 6 3 -1 7 5 6 8 -1 8 7 -1	SCC 0: 0 4 1 SCC 1: 2 SCC 2: 5 7 3 6 8 Meta-graph: 0 2 1 1 2 2