Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления» (ИУ)

КАФЕДРА «Информационная безопасность» (ИУ8)

Отчёт

по лабораторной работе № 6
по дисциплине «Теория систем и системный анализ»

Тема: «Построение сетевого графа работ и его анализ методом критического пути (CPM)»

Вариант 15

Выполнил: Ушаков З. М.,
студент группы ИУ8-31

Проверил: Коннова Н.С.,
доцент каф. ИУ8

Москва 2020

1. Цель работы

Изучить задачи сетевого планирования в управлении проектами и приобрести навыки их решения при помощи метода критического пути.

2. Условие задачи

Задан набор работ с множествами непосредственно предшествующих работ (по варианту).

1. Построить сетевой граф, произвести его топологическое упорядочение и нумерацию.

2. Рассчитать и занести в таблицу поздние сроки начала и ранние сроки окончания работ.

3. Рассчитать и занести в таблицу ранние и поздние сроки наступления событий.

4. Рассчитать полный и свободный резервы времени работ.

5. Рассчитать резерв времени событий, определить и выделить на графе критический путь.

3. Ход работы

Таблица варианта 15

№пп	P_a	P_b	P_c	P_d	P_e	P_f	P_g	P_h	P_i	P_j	P_k
15	∅	*a*	*b*	*c,g*	∅	*e*	*f,j*	*b*	*f,j*	∅	*i,h*

Реализация алгоритма нахождения критического пути сетевого графа производилось с помощью библиотеки boost/graph.

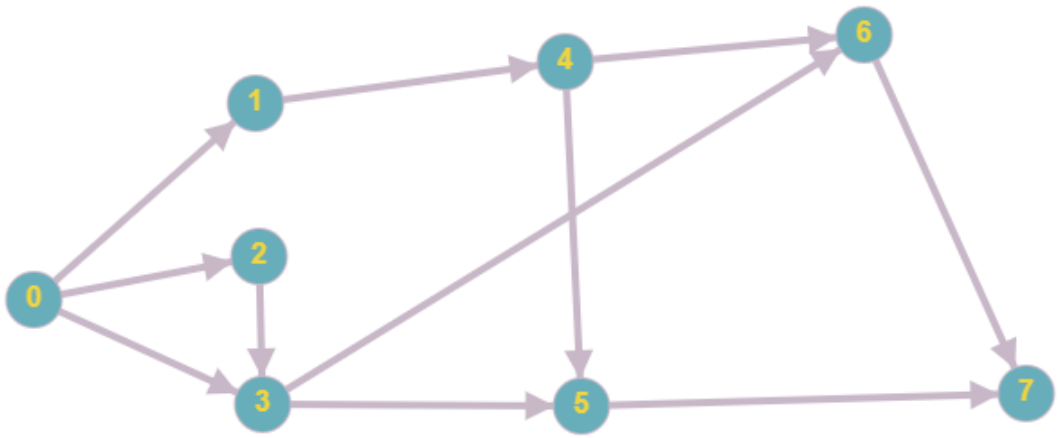
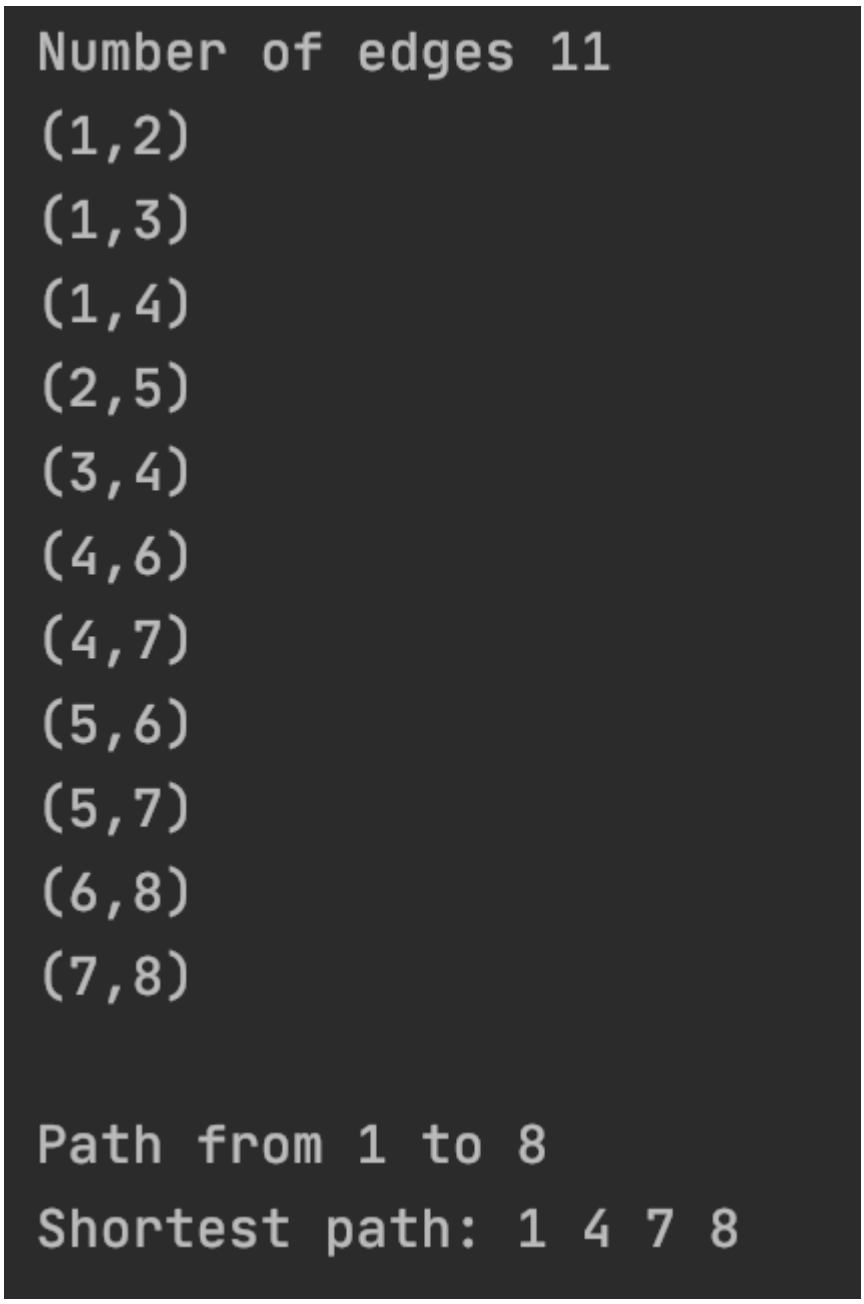


Рисунок графа

Результат работы программы представлен на следующей фотографии.



Ссылка на git-репозиторий: https://github.com/HvarZ/tsisa_lab06

4. Выводы

С помощью библиотеки boost/graph реализовал решение поставленной задачи. Все аналитические вычисления совпали с результатом работы программы.

Приложение 1. Исходный код программы

Файл Main.h:

```
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/dijkstra_shortest_paths.hpp>
#include <boost/property_map/property_map.hpp>
#include <boost/graph/graph_traits.hpp>
```

```
#include <iostream>
#include <algorithm>

using boost::add_edge;

typedef boost::property<boost::edge_weight_t, int> EdgeWeightProperty;
typedef boost::adjacency_list<boost::listS, boost::vecS, boost::directedS,
boost::no_property, EdgeWeightProperty> DirectedGraph;
typedef boost::graph_traits<DirectedGraph>::edge_iterator edge_iterator;
typedef boost::graph_traits<DirectedGraph>::vertex_descriptor vertex_descriptor;

int main() {
    DirectedGraph graph;

    int weights[] = {3, 3, 2, 5, 1, 4, 3, 2, 3, 4, 5};

    add_edge(1, 2, weights[0], graph);
    add_edge(1, 3, weights[1], graph);
    add_edge(1, 4, weights[2], graph);
    add_edge(2, 5, weights[3], graph);
    add_edge(3, 4, weights[4], graph);
    add_edge(4, 6, weights[5], graph);
    add_edge(4, 7, weights[6], graph);
    add_edge(5, 6, weights[7], graph);
    add_edge(5, 7, weights[8], graph);
    add_edge(6, 8, weights[9], graph);
    add_edge(7, 8, weights[10], graph);

    std::pair<edge_iterator, edge_iterator> ei = edges(graph);

    std::cout << "Number of edges " << num_edges(graph) << std::endl;

    std::copy(ei.first, ei.second,
std::ostream_iterator<boost::adjacency_list<>::edge_descriptor>{std::cout, "\n"});

    std::cout << std::endl;
```

```
    std::vector<vertex_descriptor> p(num_vertices(graph));
    std::vector<int> d(num_vertices(graph));

    vertex_descriptor start = vertex(1, graph);
    vertex_descriptor final = vertex(8, graph);

    boost::dijkstra_shortest_paths(graph, start,
boost::predecessor_map(&p[0]).distance_map(&d[0]));

    std::vector<boost::graph_traits<DirectedGraph>::vertex_descriptor> path;
    boost::graph_traits<DirectedGraph>::vertex_descriptor current = final;

    while(current != start) {
        path.push_back(current);
        current = p[current];
    }

    path.push_back(start);

    std::cout << "Path from " << 1 << " to " << 8 << std::endl;

std::vector<boost::graph_traits<DirectedGraph>::vertex_descriptor>::reverse_iterator
it;

    std::cout << "Shortest path: ";

    for (it = path.rbegin(); it != path.rend(); it++) {
        std::cout << *it << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Контрольный вопрос

Какие исходные данные необходимы для использования метода критического пути?
Для метода критического пути необходимы следующие данные:

- Длительность всех работ (вес каждого ребра графа)
- Множество всех предшествующих работ (каждую вершину графа)