

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ
по лабораторной работе №2
по дисциплине «Построение и анализ алгоритмов»
Тема: Алгоритмы на графах

Студент гр. 8304

Бутко А.М.

Преподаватель

Размочаева Н.В.

Санкт-Петербург

2020

Цель работы.

Реализовать алгоритм Дейкстры поиска пути в графе (на основе кода A*).

Алгоритм.

Для алгоритма Дейкстры требуется хранить множество посещенных вершин и множество необработанных вершин. Изначально расстояния до всех вершин, кроме начальной, равны бесконечности (т.к. компьютер не поддерживает бесконечность ввиду физических ограничений, то бесконечной величиной будет выступать заведомо большее число, чем цена ребра, т.е. INT_MAX). На каждой итерации из множества необработанных вершин берется вершина с минимальным расстоянием (минимальным приоритетом) и обрабатывается: происходит релаксация всех ребер, из нее исходящих, после чего вершина помещается во множество уже обработанных вершин.

Релаксация ребра в данном алгоритме: $\text{dist}[V] = \min(\text{dist}[V], \text{dist}[U] + w[U, V])$, где $\text{dist}[V]$ — расстояние от начальной вершины до вершины V, а $w[U, V]$ — вес ребра из U в V.

Реализация алгоритма.

Можем упростить хранение множеств посещенных вершин и необработанных вершин до вектора-очереди, в котором будут все необработанные вершины, а по мере работы алгоритма удаляться все обработанные вершины.

Так же была реализована структура NodeInfo, которая хранит в себе значения вершин и стоимость перемещения и текущей вершины, начальную длину и предыдущую вершину из которой был найден минимальный путь.

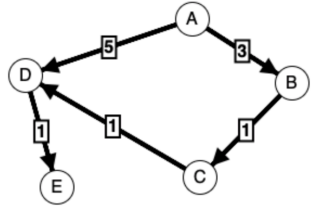
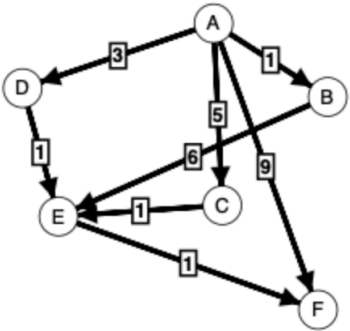
Оценка сложности алгоритма.

Каждая вершина посещается ровно один раз, следовательно требуется $O(V)$ посещений. В худшем случае, каждое перемещение по ребру приводит к изменению одного элемента структуры, в которой хранится граф, то есть, $O(E)$ изменений. Так как вершины хранятся в массиве, а для поиска минимума используется линейный поиск, временная сложность алгоритма Дейкстры: $O(V * V + E) = O(V^2)$.

Тестирование.

Тестирование приведено в таблице 1.

Таблица 1 — Тестирование программы, реализующей алгоритм Дейкстры

Input	Output	Представление графа
A E A B 3.0 B C 1.0 C D 1.0 A D 5.0 D E 1.0	ADE	
A F A B 1.0 A C 5.0 C E 1.0 B E 6.0 A D 3.0 D E 1.0 E F 1.0 A F 9.0	ADEF	

Вывод.

Алгоритм Дейкстры, а именно его сложность всецело зависит от способа реализации и хранения графа. Так же у алгоритма есть существенный недостаток — его нельзя использовать на графах с ребрами, имеющими отрицательный вес. Тем не менее алгоритм был реализован и протестирован на графе, с помощью ЯП C++.

ПРИЛОЖЕНИЕ А.

ИСХОДНЫЙ КОД.

```
#include <iostream>
#include <locale>
#include <ctime>
#include <algorithm>
#include <vector>
#include <map>

char startVertex;
char finishVertex;

struct NodeInfo
{
    std::vector<std::pair<char, double>> edges;
    double startLenght = INT_MAX;
    char parentNode;
};

void dijkstra(std::map<char, NodeInfo>& Graph)
{
    std::vector<char> Vertex;
    Vertex.reserve(Graph.size());
    for (auto& i : Graph)
        Vertex.push_back(i.first);

    while (!Vertex.empty())
    {
        char curNode = '0';
        size_t eraseInd = 0;
        double minPriority = -1;

        for(size_t i = 0; i < Vertex.size(); ++i)
        {
            double curPriority = Graph[Vertex[i]].startLenght;
            if (minPriority == -1 || curPriority < minPriority)
            {
                eraseInd = i;
                curNode = Vertex[eraseInd];
                minPriority = curPriority;
            }
        }

        if (minPriority == -1) break;

        Vertex.erase(Vertex.begin() + eraseInd);

        for (auto& childNode : Graph[curNode].edges)
        {
            double oldLenght = Graph[childNode.first].startLenght;
            double newLenght = Graph[curNode].startLenght + childNode.second;
            if (newLenght < oldLenght)
            {
                Graph[childNode.first].startLenght = newLenght;
                Graph[childNode.first].parentNode = curNode;
            }
        }
    }
}

void print(char start, char finish, std::map<char, NodeInfo>& Graph)
{
    std::string answer(1, finish);
```

```

while (true)
{
    if (answer.back() == start) break;
    answer += Graph[answer.back()].parentNode;
}

std::reverse(answer.begin(), answer.end());
std::cout << answer << std::endl;
}

int main()
{
    setlocale(LC_ALL, "Russian");

    std::map<char, NodeInfo> Graph;

    std::cout << "_____АЛГОРИТМ ДЕЙКСТРЫ_____" << std::endl;
    std::cout << "Введите начальную вершину : ";
    std::cin >> startVertex;
    std::cout << "Введите конечную вершину : ";
    std::cin >> finishVertex;

    char from = 0;
    char to = 0;
    double edge = 0;

    std::cout << "Ожидается ввод формата <откуда> <куда> <сколько>" << std::endl;
    std::cout << "Ввод закончится control + D" << std::endl;
    std::cout << "_____ " << std::endl;
    while(std::cin >> from >> to >> edge)
    {
        if (from == startVertex)
            Graph[from].startLenght = 0;
        Graph[from].edges.emplace_back(to, edge);
    }
    std::cout << "_____ " << std::endl;
    auto time = clock();
    dijkstra(Graph);
    std::cout << "Время работы алгоритма : " << (double)(clock() - time) /
CLOCKS_PER_SEC << std::endl;
    std::cout << "Сложность алгоритма для графа G(V,E) :  $O(V * V + E) \sim O(V * V)$ " <<
std::endl;
    std::cout << "Оптимальный путь : ";
    print(startVertex, finishVertex, Graph);
}

```