

ВВЕДЕНИЕ

Графы - это абстрактная структура данных, которая представляет собой набор вершин, связанных между собой ребрами. Графы широко применяются для моделирования и решения различных задач, связанных с отношениями и сетями.

Задача поиска кратчайшего пути между вершинами – одна из фундаментальных задач в теории графов. Она заключается в поиске пути между двумя вершинами графа с минимальной суммой весов ребер. Данная задача имеет множество практических применений, таких как оптимизация маршрутов в сетевых системах, планирование транспортных маршрутов, поиск оптимальных путей в графах социальных связей и других.

Существует несколько алгоритмов, которые могут быть использованы для решения задачи поиска кратчайшего пути, но одним из самых известных и широко используемых является алгоритм Дейкстры.

В данной лабораторной работе требуется изучить и реализовать алгоритмы Дейкстры для поиска кратчайшего пути на C++, адаптируя классический вариант алгоритма под представление графа матрицей смежности и списком смежности.

Для достижения поставленных целей лабораторной работы необходимо выполнить следующие задачи:

- рассмотреть и изучить представление графа матрицей смежности и списком смежности, а также разобрать алгоритм Дейкстры;
- реализовать алгоритм Дейкстры для каждого представления;
- составить блок-схемы алгоритмов;
- рассчитать трудоемкости алгоритма Дейкстры для графовых представлений;
- экспериментально проверить работу алгоритмов;
- на основании проделанной работы сделать выводы.

1 Аналитическая часть

1.1 Представление графов

Представление графа — это способ описания структуры графа с помощью данных, которые позволяют хранить информацию о вершинах и ребрах графа.

Два наиболее распространенных способа представления графа в компьютере это матрица смежности и список смежности.

Пусть задан граф, представленный на рисунке 1. Его можно представить с помощью матрицы смежности или списка смежности.

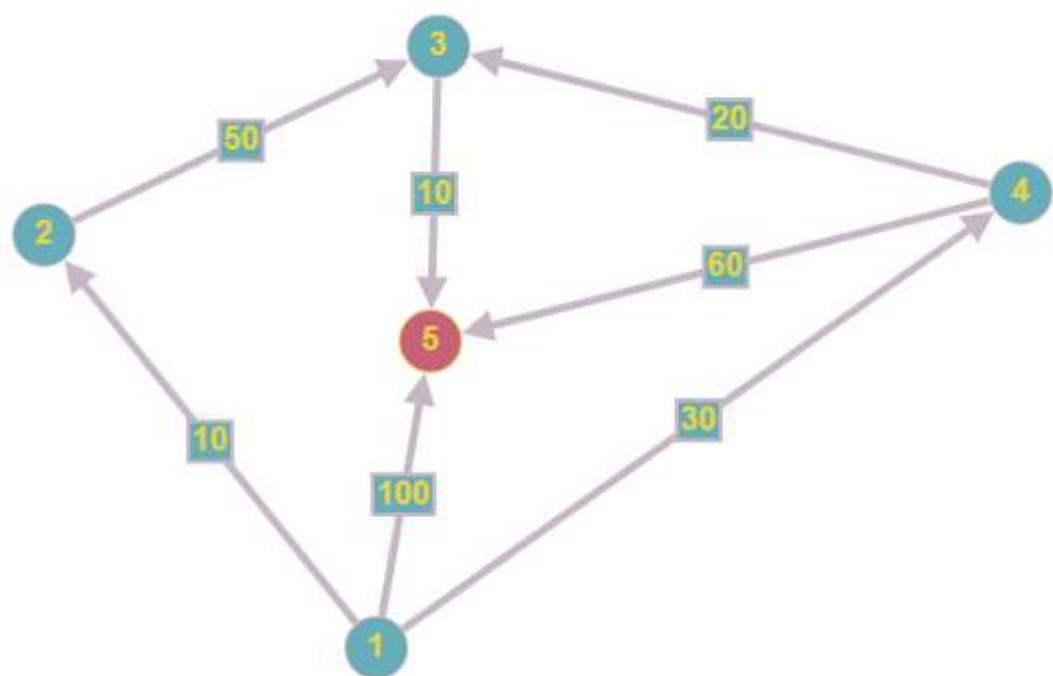


Рисунок 1 – заданный граф

Матрица смежности представляет собой граф в виде двумерного массива размером $n \times n$, где n - количество вершин в графе. Если граф является направленным, то матрица смежности будет симметричной, а

если граф неориентированный, то матрица может быть несимметричной [1, с.289].

В матрице смежности для каждой пары вершин (i, j) значение элемента $a[i][j]$ будет указывать наличие ребра между вершинами i и j . В реализации часто элементы матрицы содержат информацию о весе ребра, если граф является взвешенным. Если между вершинами нет ребра, то значение элемента может быть нулем или другим специальным обозначением. На рисунке 2 представлены матрица смежности данного графа.

0	10	0	30	100
0	0	50	0	0
0	0	0	0	10
0	0	20	0	60
0	0	0	0	0

Рисунок 2 – матрица смежности графа

Список смежности представляет граф в виде списка, где каждая вершина имеет связанный список смежных с ней вершин. Вершины могут быть представлены в виде чисел или объектов, а смежные вершины могут быть представлены списком чисел или ссылками на объекты вершин [1, с.291].

Для каждой вершины в списке смежности хранится информация о всех ее соседних вершинах. Если граф является взвешенным, то помимо списка смежных вершин хранится информация о весе ребра. На рисунке 3 представлен список смежности данного графа.

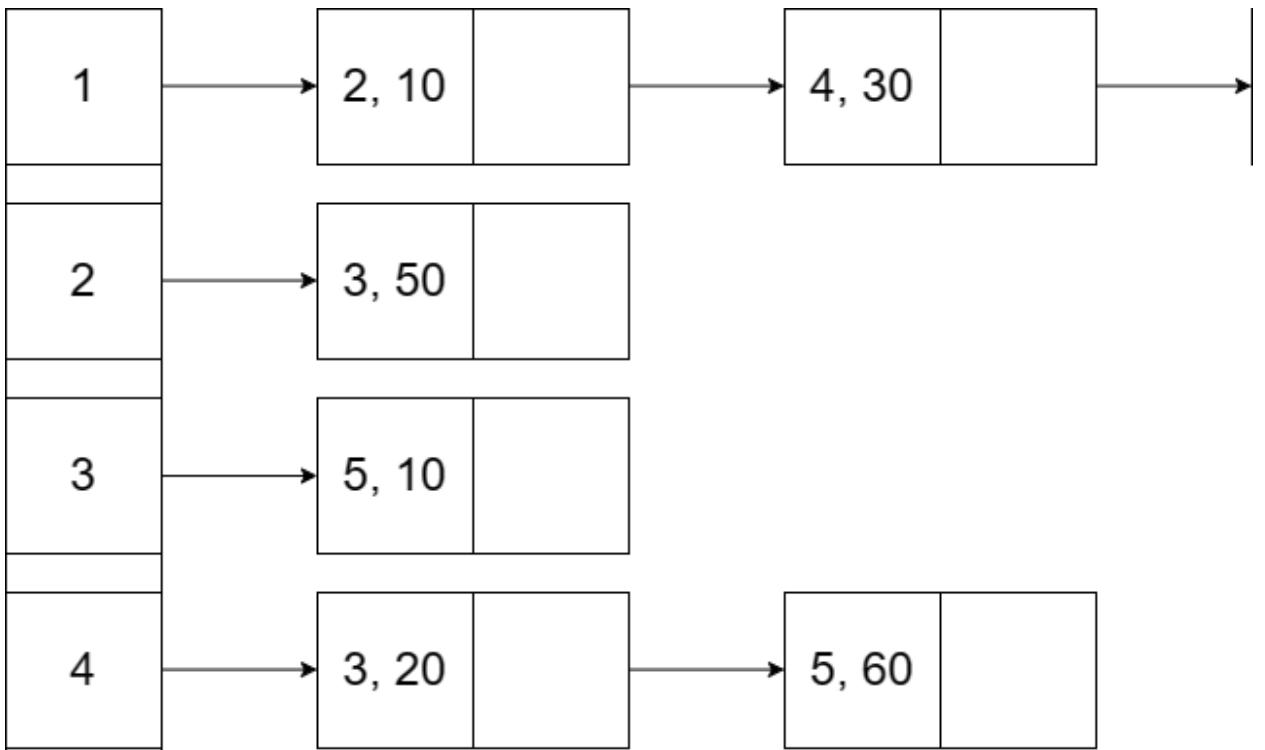


Рисунок 3 – список смежности графа

1.2 Алгоритм Дейкстры

Алгоритм Дейкстры (Dijkstra's algorithm) — это алгоритм поиска кратчайшего пути между вершинами во взвешенном графе. Он был разработан нидерландским ученым Эдсгером Дейкстрой в 1959 году. Алгоритм Дейкстры работает только с положительными весами ребер [4, с.696].

Принцип алгоритма Дейкстры заключается в нахождении оптимального пути от начальной вершины до всех остальных вершин, учитывая веса ребер графа. Алгоритм постепенно просматривает все вершины графа, обновляя их стоимости, и находит кратчайший путь до каждой из них.

Алгоритм Дейкстры гарантирует нахождение кратчайшего пути от начальной вершины до всех остальных вершин в графе, при условии, что веса ребер неотрицательны. Он является одним из самых известных алгоритмов для решения задачи поиска кратчайшего пути и

применяется в различных областях, таких как сетевое планирование, маршрутизация пакетов, GPS-навигация и другие.

Вывод

На данном этапе были изучены представления графовых структур. Кратко описаны идея и принцип работы алгоритма Дейкстры для поиска кратчайшего пути в графе.

2 Конструкторская часть

2.1 Разработка алгоритмов

С учетом аналитической части работы можно составить описательную схему алгоритма Дейкстры. Далее описаны раскрыты основные шаги алгоритма Дейкстры.

1. Инициализация: выбор начальной вершины и присваивание ей стоимости 0, а остальным вершинам присваивается бесконечная стоимость. Также создается набор не посещенных вершин.
2. Выбор вершины: на каждом шаге выбирается вершина с наименьшей стоимостью из набора не посещенных вершин. Это начальная вершина на первом шаге. После первого шага выбор вершины будет зависеть от её текущей стоимости.
3. Обновление стоимости: для каждой смежной не посещенной вершины, связанной с текущей вершиной, вычисляется новая стоимость пути до этой вершины. Новая стоимость вычисляется путем суммирования стоимости текущей вершины и веса ребра, соединяющего текущую вершину со смежной вершиной. Если полученная стоимость меньше текущей стоимости смежной вершины, она обновляется.
4. Пометка вершины: после обновления стоимостей всех смежных вершин текущей вершины, помечается текущая вершина как посещенная. Это означает, что эта вершина полностью обработана и больше не будет рассматриваться в дальнейшем.
5. Повторение: необходимо повторить шаги 2-4, пока все вершины не будут посещены. На каждом шаге выбирается вершина с наименьшей стоимостью из не посещенных вершин и обновляются стоимости смежных с ней вершин.
6. Восстановление пути: по окончании алгоритма Дейкстры получаются кратчайшие пути от начальной вершины до всех остальных вершин в графе. Чтобы восстановить сам кратчайший путь от начальной вершины до любой другой вершины, нужно начать с конечной вершины и следовать обратно по пути, выбирая вершины с минимальными стоимостями.

2.2 Трудоемкость алгоритмов

Для последующего вычисления трудоемкости алгоритмов необходимо ввести модель вычислений:

- 1) Пусть: $+, -, /, \%, ==, !=, <, >, <=, >=, [], *, ++, --$ трудоемкость 1;
- 2) Трудоемкость оператора выбора if условие then A else B рассчитывается как на рисунке 4;

$$f_{if} = f_{\text{условия}} + \begin{cases} f_A, & \text{если условие выполняется,} \\ f_B, & \text{иначе.} \end{cases}$$

Рисунок 4 - Трудоемкость оператора выбора if условие then A else B

- 3) Трудоемкость цикла for рассчитывается как на рисунке 5;

$$f_{for} = f_{\text{инициализации}} + f_{\text{сравнения}} + N(f_{\text{тела}} + f_{\text{инициализации}} + f_{\text{сравнения}})$$

Рисунок 5 - Трудоемкость оператора for

- 4) Трудоемкость вызова функции равна 0.

Произведём вычисление временной сложности алгоритма Дейкстры для разных представлений графа. Результаты представлены в таблице 1.

Таблица 1 – Расчёт временной сложности алгоритма Дейкстры для разных представлений

Представление	Трудоемкость алгоритма Дейкстры	
	Худший случай	Лучший случай
Матрица смежности	$O(V ^2)$	$O(V ^2)$
Список смежности	$O((V + E)*(\log(V)))$	$O((V + E)*(\log(V)))$

Судя по трудоемкости можно сказать, что решающую роль в значении реальной трудоемкости будет играть то, насколько граф плотный или разреженный.

Вывод

С учетом теоретических данных, полученных из аналитического раздела, разработаны блок-схемы алгоритмов Дейкстры для разных представлений графов. Кроме того, проведена теоретическая оценка их трудоемкости, которая согласуется с предполагаемыми сложностями алгоритмов.

3 Экспериментальная часть

3.1 Требования к ПО

К программе предъявляется ряд требований:

- на вход подаётся массив из целых чисел;
- на выходе — кратчайшие пути от заданной вершины до каждой из вершин графа.

Программа обрабатывает неориентированные и ориентированные графы, которые могут быть как связными, так и несвязными.

3.2 Технические характеристики

Технические характеристики устройства, на котором происходило тестирование:

- Операционная система: Windows 10 Pro 64 бит
- Память: 16 ГБ
- Процессор: Intel Core Pentium G4560 3.5 ГГц

Тестирование проводилось на ПК, включенном в сеть электропитания. Во время тестирования ПК был нагружен только встроенными приложениями окружения рабочего стола, окружением рабочего стола, а также непосредственно системой тестирования.

3.3 Тестирование программ

В таблице 2 приведены функциональные тесты для алгоритма Дейкстры, граф которого реализован матрицей смежности. Как результат – все тесты были пройдены успешно.

Таблица 2 – Тестирование алгоритма Дейкстры на матрице смежности

Входной массив	Выходной массив	Ожидаемый результат
5 0 10 0 30 100 0 0 50 0 0 0 0 0 0 10 0 0 20 0 60 0 0 0 0 0 Enter starting vertex: 0	Shortest distances from the starting vertex 0: To vertex 0: 0 To vertex 1: 10 To vertex 2: 50 To vertex 3: 30 To vertex 4: 60	Shortest distances from the starting vertex 0: To vertex 0: 0 To vertex 1: 10 To vertex 2: 50 To vertex 3: 30 To vertex 4: 60
5 0 Enter starting vertex: 3	Shortest distances from the starting vertex 3: To vertex 0: Infinity To vertex 1: Infinity To vertex 2: Infinity To vertex 3: 0 To vertex 4: Infinity	Shortest distances from the starting vertex 3: To vertex 0: Infinity To vertex 1: Infinity To vertex 2: Infinity To vertex 3: 0 To vertex 4: Infinity
5 0 0 0 0 0 0 0 0 0 0	Shortest distances from the starting vertex 4: To vertex 0: 1	Shortest distances from the starting vertex 4: To vertex 0: 1

0 0 0 0 0	To vertex 1: Infinity	To vertex 1: Infinity
0 0 0 0 0	To vertex 2: 1	To vertex 2: 1
1 0 1 1 0	To vertex 3: 1	To vertex 3: 1
Enter starting vertex: 4	To vertex 4: 0	To vertex 4: 0
{Пустой массив}	Starting vertex	Starting vertex
Enter starting vertex: 0	incorrect	incorrect

В таблице 3 приведены функциональные тесты для алгоритма Дейкстры, граф которого реализован списком смежности. Как и в случае с матрицей смежности все тесты были пройдены успешно.

Таблица 3 – Тестирование алгоритма Дейкстры на списке смежности

Входной массив	Выходной массив	Ожидаемый результат
5 0, 1, 10 0, 3, 30 0, 4, 100 1, 2, 50 2, 4, 10 3, 2, 20 3, 4, 60 4, 4, 0 0	Vertex Path from the initial vertex 0 0 1 10 2 50 3 30 4 60	Vertex Path from the initial vertex 0 0 1 10 2 50 3 30 4 60
5 0, 1, 10	Vertex Path from the initial vertex	Vertex Path from the initial vertex

0, 3, 30	0	2147483647	0	2147483647
0, 4, 100	1	2147483647	1	2147483647
1, 2, 50	2	2147483647	2	2147483647
2, 4, 10	3	2147483647	3	2147483647
3, 2, 20	4	0	4	0
3, 4, 60				
4, 4, 0				
4				
0	Empty graph!		Empty graph!	
{Пустой массив} Enter starting vertex: 0	Starting vertex incorrect		Starting vertex incorrect	

В таблице 4 приведено среднее время выполнения программ для разных представлений графов.

Таблица 4 – Время выполнения программ для разных представлений графов

V – количество вершин	E – количество дуг	Среднее время работы, мс	
		Матрица смежности	Список смежности
5	8	5	3
10	28	12	6

10	5	13	5
----	---	----	---

Вывод

По результатам эксперимента ясно, что эффективность алгоритма Дейкстры в зависимости от представления зависит от количества ребер и количества вершин. Эффективность алгоритма напрямую зависит от вида входного графа, а также от представления этого графа.

Выбор между матрицей смежности и списком смежности зависит от размера и плотности графа, а также от операций, которые требуются для конкретной задачи. Оба представления имеют свои преимущества и недостатки, и выбор оптимального способа представления зависит от контекста использования.

По результатам эксперимента можно выделить некоторые особенности использования списка смежности:

1. Довольно эффективно работает с большими графами и разреженными графиками;
2. Занимает меньше памяти для хранения разреженных графов;
3. Проще добавлять или удалять вершины и ребра;
4. Затруднительно проверить наличие ребра между двумя вершинами;
5. Обход всех соседних вершин может быть медленнее по сравнению с матрицей смежности.

Аналогично можно выделить особенности матрицы смежности:

1. Легко проверить наличие ребра между двумя вершинами;
2. Эффективно работает с небольшими графами и с графиками с большим количеством ребер;
3. Требует больше памяти для хранения, чем список смежности, особенно для графов с малым количеством ребер.

ЗАКЛЮЧЕНИЕ

В процессе выполнения лабораторной работы были проведены анализ, подробное изучение, реализация и тестирование алгоритма Дейкстры для представлений графа матрицей смежности и списком смежности, были выполнены следующие задачи:

- рассмотрены и изучены представления графа матрицей смежности и списком смежности, а также разобран алгоритм Дейкстры;
- реализован алгоритм Дейкстры для каждого представления;
- составлены блок-схемы алгоритмов;
- рассчитаны трудоемкости алгоритма Дейкстры для двух графовых представлений;
- экспериментально проверена работа алгоритмов;
- на основании проделанной работы сделаны выводы.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Белоусов А.И., Ткачев С.Б. – Дискретная математика, 2015. – 288 с.
2. Роберт Седжвик. Алгоритмы на C++. Фундаментальные алгоритмы и структуры данных. Algorithms in C++. — М.: «Вильямс», 2011. — 189 с.
3. Полный справочник по C++, 4-е издание C++: The Complete Reference, 4th Edition. — М.: «Вильямс», 2011. — 99 с.
4. Т.Кормен, Ч.Лейзерсон, Р.Ривест, К.Штайн – Алгоритмы. Построение и анализ, 2013. – 680 с.
5. С.Пратта. Язык программирования C++. Лекции и упражнения, 2012. – 79 с.

ПРИЛОЖЕНИЕ А

Листинг программы

Листинг 1 – Алгоритм Дейкстры при представлении графа матрицей смежности

```
#include <iostream>
#include <fstream>
#include <limits>

#define INF std::numeric_limits<int>::max()

void dijkstra(int** graph, int vertices, int start)
{
    int* distances = new int[vertices];
    bool* visited = new bool[vertices];

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

    distances[start] = 0;

    bool hasPath = true;

    for (int count = 0; count < vertices - 1; ++count)
    {
        int minDistance = INF;
        int minIndex = -1;

        for (int v = 0; v < vertices; ++v)
        {
            if (!visited[v] && distances[v] < minDistance)
            {
                minDistance = distances[v];
                minIndex = v;
            }
        }

        visited[minIndex] = true;

        if (minIndex == -1)
        {
            //Если не найдено непосещенных вершин, выходим из цикла
            break;
        }

        for (int v = 0; v < vertices; ++v)
        {
            if (!visited[v] && graph[minIndex][v] && distances[minIndex] !=
INF &&
                distances[minIndex] + graph[minIndex][v] < distances[v])
            {

```

```

            distances[v] = distances[minIndex] + graph[minIndex][v];
        }
    }

    if (distances[minIndex] == INF)
    {
        hasPath = false;
    }
}

std::cout << "Shortest distances from the starting vertex " << start <<
":\n";
for (int i = 0; i < vertices; ++i)
{
    if (distances[i] == INF)
    {
        std::cout << "To vertex " << i << ":"; //i+1 для удобного
        сравнения с graphonline
        if (!hasPath)
        {
            std::cout << "No way\n";
        }
        else
        {
            std::cout << "Infinity\n";
        }
    }
    else
    {
        std::cout << "To vertex " << i << ":" << distances[i] << "\n";
//i+1 для удобного сравнения с graphonline
    }
}

delete[] distances;
delete[] visited;
}

```

```

int main()
{
    std::ifstream inputFile("graph.txt");
    if (!inputFile)
    {
        std::cout << "Impossible to open file";
        return 1;
    }

    int vertices;
    inputFile >> vertices;
    inputFile.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
//Пропускаем пустую строку

    int** graph = new int*[vertices];
    for (int i = 0; i < vertices; ++i)
    {
        graph[i] = new int[vertices];
        for (int j = 0; j < vertices; ++j)
        {
            inputFile >> graph[i][j];
        }
    }
}

```

```

    inputFile.close(); //Закрываем файл после чтения данных

    int startVertex;

    std::cout << "Enter starting vertex: "; //ввод с 0
    std::cin >> startVertex;

    if (startVertex >= 0 && startVertex < vertices)
    {
        dijkstra(graph, vertices, startVertex);
    }
    else
    {
        std::cout << "Starting vertex incorrect.\n";
    }

    for (int i = 0; i < vertices; ++i)
    {
        delete[] graph[i];
    }
    delete[] graph;

    return 0;
}

```

Листинг 2 – Алгоритм Дейкстры при представлении графа списком смежности

```

#include <stdio.h>
#include <iostream>
#include <stdlib.h>
#include <limits.h>

//Алгоритм Дейкстры с графом, представленным списком смежности

//Структура для представления узла в списке смежности
struct AdjListNode
{
    int dest;
    int weight;
    int isDirected; //0 - неориентированный граф, 1 - ориентированный граф
    struct AdjListNode* next;
};

//Структура для представления списка смежности
struct AdjList
{
    //Указатель на голову списка
    struct AdjListNode* head;
};

//Структура для представления графа
//Граф представляет собой массив списков смежности
//Размер массива будет V количество вершин в графе
struct Graph
{
    int V;
    struct AdjList* array;
}

```

```

};

//Функция для создания нового узла списка смежности
struct AdjListNode* newAdjListNode(int dest, int weight)
{
    struct AdjListNode* newNode = (struct AdjListNode*)malloc(sizeof(struct
AdjListNode));
    newNode->dest = dest;
    newNode->weight = weight;
    newNode->isDirected = 0; //По умолчанию неориентированное ребро
    newNode->next = NULL;
    return newNode;
}

//Функция для создания графа с V вершинами
struct Graph* createGraph(int V)
{
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->V = V;

    //Создание массива списков смежности.
    //Размер массива будет V
    graph->array = (struct AdjList*)malloc(V * sizeof(struct AdjList));

    //Инициализация каждого списка смежности как пустого путем установки
    головы в NULL
    for (int i = 0; i < V; ++i)
        graph->array[i].head = NULL;

    return graph;
}

//Добавление ребра в ориентированный или неориентированный граф
void addEdge(struct Graph* graph, int src, int dest, int weight)
{
    //Добавление ребра от src к dest.
    //Новый узел добавляется в начало списка смежности src.
    struct AdjListNode* newNode = newAdjListNode(dest, weight);
    newNode->isDirected = 1; //Ориентированное ребро
    newNode->next = graph->array[src].head;
    graph->array[src].head = newNode;

    //Если граф неориентированный, добавить ребро от dest к src
    if (!newNode->isDirected)
    {
        newNode = newAdjListNode(src, weight);
        newNode->next = graph->array[dest].head;
        graph->array[dest].head = newNode;
    }
}

//Структура для представления узла мин-кучи
struct MinHeapNode
{
    int v;
    int dist;
};

//Структура для представления мин-кучи
struct MinHeap
{
    //Количество узлов кучи
    int size;
}

```

```

//Емкость мин-кучи
int capacity;
//Это нужно для функции decreaseKey()
int *pos;
struct MinHeapNode **array;
};

//Функция для создания нового узла мин-кучи
struct MinHeapNode* newMinHeapNode(int v, int dist)
{
    struct MinHeapNode* minHeapNode = (struct
MinHeapNode*)malloc(sizeof(struct MinHeapNode));
    minHeapNode->v = v;
    minHeapNode->dist = dist;
    return minHeapNode;
}

//Функция для создания мин-кучи
struct MinHeap* createMinHeap(int capacity)
{
    struct MinHeap* minHeap = (struct MinHeap*)malloc(sizeof(struct
MinHeap));
    minHeap->pos = (int *)malloc(capacity * sizeof(int));
    minHeap->size = 0;
    minHeap->capacity = capacity;
    minHeap->array = (struct MinHeapNode**)malloc(capacity * sizeof(struct
MinHeapNode*));
    return minHeap;
}

//Функция для обмена двух узлов мин-кучи.
//Необходимо для функции minHeapify
void swapMinHeapNode(struct MinHeapNode** a, struct MinHeapNode** b)
{
    struct MinHeapNode* t = *a;
    *a = *b;
    *b = t;
}

//Стандартная функция для преобразования кучи в мин-кучу при заданном индексе
//Эта функция плюсом обновляет позицию узлов при их обмене.
//Позиция необходима для функции decreaseKey()
void minHeapify(struct MinHeap* minHeap, int idx)
{
    int smallest, left, right;
    smallest = idx;
    left = 2 * idx + 1;
    right = 2 * idx + 2;

    if (left < minHeap->size &&
        minHeap->array[left]->dist < minHeap->array[smallest]->dist)
        smallest = left;

    if (right < minHeap->size &&
        minHeap->array[right]->dist < minHeap->array[smallest]->dist)
        smallest = right;

    if (smallest != idx)
    {
        //Узлы, которые нужно обменять в мин-куче
        struct MinHeapNode* smallestNode = minHeap->array[smallest];
        struct MinHeapNode* idxNode = minHeap->array[idx];

```

```

    //Обмен позициями
    minHeap->pos[smallestNode->v] = idx;
    minHeap->pos[idxNode->v] = smallest;

    //Обмен узлами
    swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);

    minHeapify(minHeap, smallest);
}
}

//Вспомогательная функция для проверки, пустая ли данная мин-куча или нет
int isEmpty(struct MinHeap* minHeap)
{
    return minHeap->size == 0;
}

//Стандартная функция для извлечения минимального узла из кучи
struct MinHeapNode* extractMin(struct MinHeap* minHeap)
{
    if (isEmpty(minHeap))
        return NULL;

    //Сохранение корневого узла
    struct MinHeapNode* root = minHeap->array[0];

    //Замена корневого узла последним узлом
    struct MinHeapNode* lastNode = minHeap->array[minHeap->size - 1];
    minHeap->array[0] = lastNode;

    //Обновление позиции последнего узла
    minHeap->pos[root->v] = minHeap->size - 1;
    minHeap->pos[lastNode->v] = 0;

    //Уменьшение размера кучи и преобразование корня
    --minHeap->size;
    minHeapify(minHeap, 0);

    return root;
}

//Функция для уменьшения значения dist у заданной вершины v. Эта функция
//использует pos[] мин-кучи для получения текущего индекса узла в мин-куче.
void decreaseKey(struct MinHeap* minHeap, int v, int dist)
{
    //Получение индекса v в массиве кучи
    int i = minHeap->pos[v];

    //Получение узла и обновление его значения dist
    minHeap->array[i]->dist = dist;

    //Проход вверх по дереву, пока не будет сформирована полная мин-куча.
    //Это цикл с O(Logn) итерациями
    while (i && minHeap->array[i]->dist < minHeap->array[(i - 1) / 2]->dist)
    {
        //Обмен этого узла с его родителем
        minHeap->pos[minHeap->array[i]->v] = (i - 1) / 2;
        minHeap->pos[minHeap->array[(i - 1) / 2]->v] = i;
        swapMinHeapNode(&minHeap->array[i], &minHeap->array[(i - 1) / 2]);

        //Переход к индексу родителя
        i = (i - 1) / 2;
    }
}

```

```

}

//Вспомогательная функция для проверки, находится ли заданная вершина 'v'
//в мин-куче или нет
bool isInMinHeap(struct MinHeap* minHeap, int v)
{
    if (minHeap->pos[v] < minHeap->size)
        return true;
    return false;
}

//Вспомогательная функция для печати решения
void printArr(int dist[], int n)
{
    printf("Vertex | Path from the initial vertex\n");
    for (int i = 0; i < n; ++i)
        printf("%d \t\t %d\n", i, dist[i]);
}

//Основная функция, которая вычисляет кратчайшие пути от источника 'src' ко
//всем
//вершинам. Сложность функции O(ELogV)
void dijkstra(struct Graph* graph, int src)
{
    int V = graph->V; //Получение количества вершин в графе
    int dist[V]; //Значения расстояний, используемые для выбора ребра
    минимального веса в разрезе

    //minHeap представляет множество E
    struct MinHeap* minHeap = createMinHeap(V);

    //Инициализация мин-кучи со всеми вершинами и расстояниями равными
    INT_MAX
    for (int v = 0; v < V; ++v)
    {
        dist[v] = INT_MAX;
        minHeap->array[v] = newMinHeapNode(v, dist[v]);
        minHeap->pos[v] = v;
    }

    //Установка расстояния для исходной вершины в 0,
    //так как расстояние от исходной вершины до себя равно 0
    minHeap->array[src] = newMinHeapNode(src, dist[src]);
    minHeap->pos[src] = src;
    dist[src] = 0;
    decreaseKey(minHeap, src, dist[src]);

    //Размер мин-кучи равен V
    minHeap->size = V;

    //В конечном итоге размер minHeap станет равен 0
    while (!isEmpty(minHeap))
    {
        //Извлечение вершины с минимальным расстоянием из кучи
        struct MinHeapNode* minHeapNode = extractMin(minHeap);
        int u = minHeapNode->v; //Сохранение извлеченной вершины
        struct AdjListNode* pCrawl = graph->array[u].head;

        //Перебор всех смежных вершин данной извлеченной вершины (взвешенные
ребра)
        while (pCrawl != NULL)
        {
            int v = pCrawl->dest;

```

```

        //Если v находится в мин-куче и его расстояние больше нового
        //расстояния u + weight
        if (isInMinHeap(minHeap, v) && dist[u] != INT_MAX && pCrawl-
>weight + dist[u] < dist[v])
        {
            dist[v] = dist[u] + pCrawl->weight;

            //Обновление расстояния v в мин-куче
            decreaseKey(minHeap, v, dist[v]);
        }
        pCrawl = pCrawl->next;
    }

    //Печать рассчитанных расстояний
    printArr(dist, V);
}

//Задание графа
int main()
{
    //Создание графа
    int V = 5; //Число вершин
    if (V != 0) {
        struct Graph* graph = createGraph(V);

        //Добавление ребер
        addEdge(graph, 0, 1, 10);
        addEdge(graph, 0, 3, 30);
        addEdge(graph, 0, 4, 100);
        addEdge(graph, 1, 2, 50);
        addEdge(graph, 2, 4, 10);
        addEdge(graph, 3, 2, 20);
        addEdge(graph, 3, 4, 60);
        addEdge(graph, 4, 4, 0);

        dijkstra(graph, 4);
    }
    else {
        std::cout << "Empty graph!";
    }

    return 0;
}

```