Лабораторная работа № 10. Определение минимального пути на графе

**Цель работы:** изучить алгоритмы Дейкстры и Флойда-Уоршелла для определения минимального пути на графе.

### Краткие теоретические сведения

**Алгоритмы нахождения кратчайшего пути**

Нахождение *кратчайшего пути* на сегодняшний день является жизненно необходимой задачей и используется практически везде, начиная от нахождения оптимального маршрута между двумя объектами на местности (например, *кратчайший путь* от дома до университета), в системах автопилота, для нахождения оптимального маршрута при перевозках,  *коммутации*  информационного пакета в сетях и т.п.

*Поиск* *кратчайшего пути* ведется между двумя заданными вершинами в графе. Результатом является *путь*, то есть последовательность вершин и ребер, *инцидентных* двум соседним вершинам, и его *длина*.

Наиболее *эффективными алгоритмами* нахождения *кратчайшего пут являются* :

* *алгоритм Дейкстры*;
* *алгоритм Флойда*;
* переборные алгоритмы.

Указанные алгоритмы легко выполняются при малом количестве вершин в графе. При увеличении их количества задача поиска *кратчайшего пути* усложняется.

1. **Алгоритм Дейкстры**

Данный *алгоритм* является алгоритмом на графах, который изобретен нидерландским ученым Э. Дейкстрой в 1959 году. *Алгоритм* находит кратчайшее *расстояние* от одной из *вершин графа* до всех остальных и работает только для графов без ребер отрицательного веса.

Каждой вершине приписывается *вес* – это *вес* пути от начальной вершины до данной. Также каждая *вершина* может быть выделена. Если *вершина* выделена, то *путь* от нее до начальной вершины кратчайший, если нет – то временный. *Обходя граф*, *алгоритм* считает для каждой вершины *маршрут*, и, если он оказывается кратчайшим, выделяет вершину. Весом данной вершины становится *вес* пути. Для всех соседей данной вершины *алгоритм* также рассчитывает *вес*, при этом ни при каких условиях не выделяя их. *Алгоритм* заканчивает свою работу, дойдя до конечной вершины, и весом *кратчайшего пути* становится *вес* конечной вершины.

***Алгоритм Дейкстры***

Шаг 1. Всем вершинам, за исключением первой, присваивается *вес* равный бесконечности, а первой вершине – 0.

Шаг 2. Все вершины не выделены.

Шаг 3. Первая *вершина* объявляется текущей.

Шаг 4. *Вес* всех невыделенных вершин пересчитывается по формуле: *вес* невыделенной вершины есть минимальное число из старого веса данной вершины и суммы веса текущей вершины и веса *ребра*, соединяющего текущую вершину с невыделенной.

Wi = min(Wiold, Wcur + Wedcur)

Шаг 5. Среди невыделенных вершин ищется *вершина* с минимальным весом. Если таковая не найдена, то есть *вес* всех вершин равен бесконечности, то *маршрут* не существует. Следовательно, *выход*. Иначе, текущей становится найденная *вершина*. Она же выделяется.

Шаг 6. Если текущей вершиной оказывается конечная, то *путь* найден, и его *вес* есть *вес* конечной вершины иначе. Переход на шаг 4.

//////////////////////////////////////

10 шагов для написания кода алгоритма Дейкстры:

1. Создаем набор данных посещено для всех узлов графа.

2. Создаем набор данных расстояние для всех узлов графа.

3. Присваиваем посещено значение false (не посещено), а расстояние делаем равным бесконечности (недостижимо).

4. Выбираем начальный узел и присваиваем расстоянию этого узла значение ноль.

5. В первой итерации начальным узлом будет текущий узел.

6. Теперь текущий узел прокладывает путь до всех соседних узлов и просчитывает предварительное расстояние.

7. Предварительное расстояние — это путь от текущего узла до грани соседнего узла.

8. Если предварительное расстояние меньше заданного, то расстояние обновляется.

9. Меняем текущий узел, задавая значение true в посещено.

10. Выбираем следующий текущий узел с наименьшим расстоянием и значением false в посещено.

11.Повторяем шаги 6–9 то тех пор, пока все узлы не окажутся посещенными, или во всех оставшихся не посещенных узлах расстояние не изменится на бесконечность (недостижимо).

////////////////////////////////

Для программной реализации алгоритма понадобиться два массива: логический visited – для хранения информации о посещенных вершинах и численный distance, в который будут заноситься найденные кратчайшие пути.

Итак, имеется граф G=(V, E). Каждая из вершин входящих во множество V, изначально отмечена как не посещенная, т. е. элементам массива visited присвоено значение false. Поскольку самые выгодные пути только предстоит найти, в каждый элемент вектора distance записывается такое число, которое заведомо больше любого потенциального пути (обычно это число называют бесконечностью, но в программе используют, например максимальное значение конкретного типа данных). В качестве исходного пункта выбирается вершина s и ей приписывается нулевой путь: distance[s]=0, т. к. нет ребра из s в s (метод не предусматривает петель). Далее, находятся все соседние вершины (в которые есть ребро из s) [пусть таковыми будут t и u] и поочередно исследуются, а именно вычисляется стоимость маршрута из s поочередно в каждую из них:

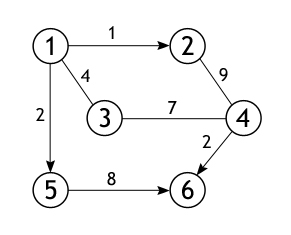
* distance[t]=distance[s]+min(told, d(s,t)), told – текущий вес вершины t, d(s,t) -вес инцидентного s и t ребра;

аналогично distance[u]=distance[s]+ min(uold, d(s,u)).

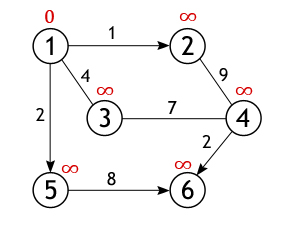
Но вполне вероятно, что в ту или иную вершину из s существует несколько путей, поэтому цену пути в такую вершину в массиве distance придется пересматривать, тогда наибольшее (неоптимальное) значение игнорируется, а наименьшее ставится в соответствие вершине.

После обработки смежных с s вершин она помечается как посещенная: visited[s]=true, и активной становится та вершина, путь из s в которую минимален. Допустим, путь из s в u короче, чем из s в t, следовательно, вершина u становиться активной и выше описанным образом исследуются ее соседи, за исключением вершины s. Далее, u помечается как пройденная: visited[u]=true, активной становится вершина t, и вся процедура повторяется для нее. Алгоритм Дейкстры продолжается до тех пор, пока все доступные из s вершины не будут исследованы.

Пример. На конкретном неориентированном графе проследим работу алгоритма, найдем все кратчайшие пути между исходной и всеми остальными вершинами. Размер (количество ребер) изображенного ниже графа равен 7 (|E|=7), а порядок (количество вершин) – 6 (|V|=6). Это взвешенный граф, каждому из его ребер поставлено в соответствие некоторое числовое значение, поэтому ценность маршрута необязательно определяется числом ребер, лежащих между парой вершин.

[](http://kvodo.ru/wp-content/uploads/algoritmh_dijkstra.jpg)

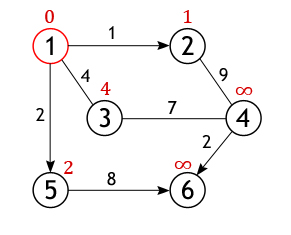
Из всех вершин входящих во множество V выберем одну, ту, от которой необходимо найти кратчайшие пути до остальных доступных вершин. Пусть таковой будет вершина 1. Длина пути до всех вершин, кроме первой, изначально равна бесконечности, а до нее – 0, т. к. граф не имеет петель.

[](http://kvodo.ru/wp-content/uploads/algoritmh_dijkstra1.jpg)

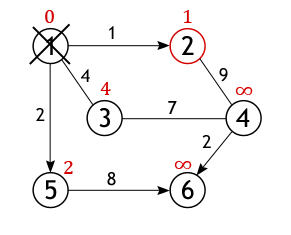
У вершины 1 ровно 3 соседа (вершины 2, 3, 5), и чтобы вычислить длину пути до них нужно сложить вес дуг, лежащих между вершинами: 1 и 2, 1 и 3, 1 и 5 со значением первой вершины (с нулем):

* 2←1+0
* 3←4+0
* 5←2+0

Как уже отмечалось, получившиеся значения присваиваются вершинам, лишь в том случае если они «лучше» (меньше) тех, которые значатся на настоящий момент. А так как каждое из трех чисел меньше бесконечности, они становятся новыми величинами, определяющими длину пути из вершины 1 до вершин 2, 3 и 5.

[](http://kvodo.ru/wp-content/uploads/algoritmh_dijkstra2.jpg)

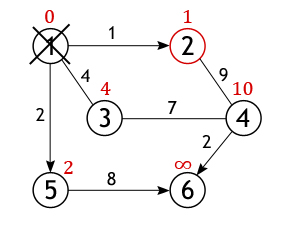
Далее, активная вершина помечается как посещенная, статус «активной» (красный круг) переходит к одной из ее соседок, а именно к вершине 2, поскольку она имеет минимальный вес.

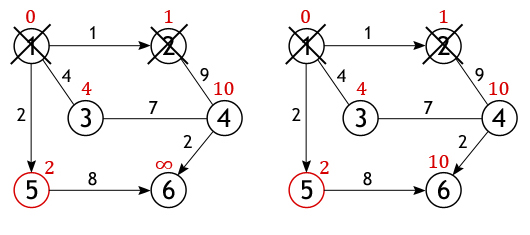
[](http://kvodo.ru/wp-content/uploads/algoritmh_dijkstra3.jpg)

У вершины 2 всего один не рассмотренный сосед (вершина 1 помечена как посещенная), расстояние до которого из нее равно 9, но нам необходимо вычислить длину пути из истоковой вершины, для чего нужно сложить величину, приписанную вершине 2 с весом дуги из нее в вершину 4:

* 4←1+9

Условие «краткости» (10<∞) выполняется, следовательно, вершина 4 получает новое значение длины пути.

[](http://kvodo.ru/wp-content/uploads/algoritmh_dijkstra4.jpg)

Вершина 2 перестает быть активной, также, как и вершина 1, удаляется из списка не посещённых. Теперь тем же способом исследуются соседи вершины 5, и вычисляется расстояние до них.[](http://kvodo.ru/wp-content/uploads/algoritmh_dijkstra11.jpg)

Когда дело доходит до осмотра соседей вершины 3, то тут важно не ошибиться, т. к. вершина 4 уже была исследована и расстояние одного из возможных путей из истока до нее вычислено. Если двигаться в нее через вершину 3, то путь составит 4+7=11, а 11>10, поэтому новое значение игнорируется, старое остается.



Аналогичная ситуация с вершиной 6. Значение самого близкого пути до нее из вершины 1 равно 10, а оно получается только в том случае, если идти через вершину 5.

Когда все вершины графа, либо все те, что доступны из истока, будут помечены как посещенные, тогда работа алгоритма Дейкстры завершится, и все найденные пути будут кратчайшими. Так, например, будет выглядеть список минимальных расстояний лежащих между вершиной 1 и всеми остальными вершинами, рассматриваемого графа:

1→1=0  
1→2=1  
1→3=4  
1→4=10  
1→5=2  
1→6=10

В программе, находящей ближайшие пути между вершинами посредством метода Дейкстры, граф будет представлен в виде не бинарной матрицы смежности. Вместо единиц в ней будут выставлены веса ребер, функция нулей останется прежней: показывать, между какими вершинами нет ребер или же они есть, но отрицательно направлены.

Реализация алгоритма Дейкстры на C++:

*#include "stdafx.h"*  
*#include <iostream>*  
using namespace std;  
const int V=6;  
*//алгоритм Дейкстры*  
void Dijkstra(int GR[V][V], int st)  
{  
int distance[V], count, index, i, u, m=st+1;  
bool visited[V];  
for (i=0; i<V; i++)  
{  
distance[i]=INT\_MAX; visited[i]=false;  
}  
distance[st]=0;  
for (count=0; count<V-1; count++)  
{  
int min=INT\_MAX;  
for (i=0; i<V; i++)  
if (!visited[i] && distance[i]<=min)  
{  
min=distance[i]; index=i;  
}  
u=index;  
visited[u]=true;  
for (i=0; i<V; i++)  
if (!visited[i] && GR[u][i] && distance[u]!=INT\_MAX &&  
distance[u]+GR[u][i]<distance[i])  
distance[i]=distance[u]+GR[u][i];  
}

cout<<"Стоимость пути из начальной вершины до остальных:\t\n";  
for (i=0; i<V; i++) if (distance[i]!=INT\_MAX)  
cout<<m<<" > "<<i+1<<" = "<<distance[i]<<endl;  
else cout<<m<<" > "<<i+1<<" = "<<"маршрут недоступен"<<endl;  
}  
*//главная функция*  
void main()  
{  
setlocale(LC\_ALL, "Rus");

int start, GR[V][V]={  
{0, 1, 4, 0, 2, 0},  
{0, 0, 0, 9, 0, 0},  
{4, 0, 0, 7, 0, 0},  
{0, 9, 7, 0, 0, 2},  
{0, 0, 0, 0, 0, 8},  
{0, 0, 0, 0, 0, 0}};  
cout<<"Начальная вершина >> "; cin>>start;  
Dijkstra(GR, start-1);  
system("pause>>void");

}

Сложность *алгоритма Дейкстры* зависит от способа нахождения вершины, а также способа хранения *множества* непосещенных вершин и способа обновления длин.

Если для представления графа использовать *матрицу смежности*, то *время выполнения* этого алгоритма имеет порядок O(n2), где n – количество *вершин графа*.

1. **Алгоритм Флойда – Уоршелла**

Алгоритм получил название в честь двух американских исследователей Роберта Флойда и Стивена Уоршелла, одновременно открывших его в 1962 году.

Алгоритм Флойда – Уоршелла – динамический алгоритм вычисления значений кратчайших путей для каждой из вершин графа. Метод работает на взвешенных графах, с положительными и отрицательными весами ребер, но без отрицательных циклов, являясь, таким образом, более общим в сравнении с алгоритмом Дейкстры, т. к. последний не работает с отрицательными весами ребер, и к тому же классическая его реализация подразумевает определение оптимальных расстояний от одной вершины до всех остальных. (между всеми вершинами)

Для реализации алгоритма Флойда – Уоршелла сформируем матрицу смежности D[][] графа G=(V, E), в котором каждая вершина пронумерована от 1 до |V|. Эта матрица имеет размер |V|´|V|, и каждому ее элементу D[i][j] присвоен вес ребра, идущего из вершины i в вершину j. По мере выполнения алгоритма, данная матрица будет перезаписываться: в каждую из ее ячеек внесется значение, определяющее оптимальную длину пути из вершины i в вершину j (отказ от выделения специального массива для этой цели сохранит память и время). Теперь, перед составлением основной части алгоритма, необходимо разобраться с содержанием матрицы кратчайших путей. Поскольку каждый ее элемент D[i][j] должен содержать наименьший из имеющихся маршрутов, то сразу можно сказать, что для единичной вершины он равен нулю, даже если она имеет петлю (отрицательные циклы не рассматриваются), следовательно, все элементы главной диагонали (D[i][i]) нужно обнулить. А чтобы нулевые недиагональные элементы (матрица смежности могла иметь нули в тех местах, где нет непосредственного ребра между вершинами i и j) сменили по возможности свое значение, определим их равными бесконечности, которая в программе может являться, например, максимально возможной длинной пути в графе, либо просто – большим числом.

Ключевая часть алгоритма, состоит из трех циклов, выражения и условного оператора, записывается довольно компактно:

**For (k=1; k<= |V|; k++)  
 For(i=1; i<= |V|; i++)**

**For(j=1; j<= |V|; j++)**

**IF (D[i][k]+D[k][j]) < D[i][j]**

**D[i][j] =D[i][k]+D[k][j]**

Кратчайший путь из вершины i в вершину j может проходить, как через них самих, так и через множество других вершин k∈(1, …, |V|). Оптимальным из i в j будет путь, имеющий минимальный вес. Этим путем может быть как прямой путь из i в j (ребро, соединяющее вершины i и j), так и путь, проходящий через промежуточные. Если путь через вершинe k меньше прямого пути, то этот путь записывается в матрицу D в качестве значения D[i][j].

Код программы на C++:

*#include "stdafx.h"*  
*#include <iostream>*  
using namespace std;  
const int maxV=1000;  
int i, j, n;  
int GR[maxV][maxV];  
*//алгоритм Флойда-Уоршелла*  
void FU(int D[][maxV], int V)  
{  
int k;  
for (i=0; i<V; i++) D[i][i]=0;  
  
for (k=0; k<V; k++)  
for (i=0; i<V; i++)  
for (j=0; j<V; j++)  
if (D[i][k] && D[k][j] && i!=j)  
if (D[i][k]+D[k][j]<D[i][j] || D[i][j]==0)  
D[i][j]=D[i][k]+D[k][j];  
  
for (i=0; i<V; i++)  
for (j=0; j<V; j++) cout<<D[i][j]<<"\t";  
cout<<endl;  
}  
}  
*//главная функция*  
void main()  
{  
setlocale(LC\_ALL, "Rus");  
cout<<"Количество вершин в графе > "; cin>>n;  
cout<<"Введите матрицу весов ребер:\n";  
for (i=0; i<n; i++)  
for (j=0; j<n; j++)  
{  
cout<<"GR["<<i+1<<"]["<<j+1<<"] > ";  
cin>>GR[i][j];  
}  
cout<<"Матрица кратчайших путей:"<<endl;  
FU(GR, n);  
system("pause>>void");  
}

Пример. Рассматриваемый граф представлен на рис 1. Количество вершин в графе 3.

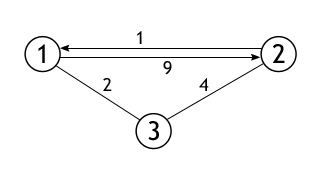
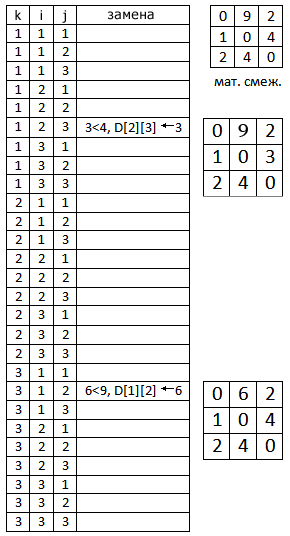
[](http://kvodo.ru/wp-content/uploads/floyd_uorshell.jpg)

Рисунок 1. Граф

Матрица смежности этого графа имеет вид :

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  | 1 | 2 | 3 |
| 1 |  |  | 0 | 9 | 2 |
| 2 |  |  | 1 | 0 | 4 |
| 3 |  |  | 2 | 4 | 0 |

Задача алгоритма: перезаписать данную матрицу так, чтобы каждая ячейка вместо веса ребра из i в j, содержала кратчайший путь из i в j.

[](http://kvodo.ru/wp-content/uploads/Floyd-Warshall-algorithm.png)

В данной таблице показаны 27 шагов выполнения основной части алгоритма. Их столько по той причине, что время выполнения метода равно O(|V|3). Первая замена происходит на итерации, при которой k=1, i=2, а j=3 (из 2 в 3 через 1). В тот момент D[2][1]=1, D[1][3]=2, D[2][3]=4. Условие истинно, т. е. D[1][3]+D[3][2]=3, а 3<4, следовательно, элемент матрицы D[2][3] получает новое значение. Следующий шаг, когда условие также истинно привносит изменения в элемент, расположенный на пересечении второй строки и третьего столбца.

Сложность O(n3)

1. Порядок выполнения работы
2. Выбрать по номеру студента в журнале свой вариант задания.
3. По списку дуг с указанием их длин составить рисунок ориентированного графа.
4. Найти для этого графа наименьший путь от вершины-входа (выбирается произвольно) до всех остальных вершин.

**Задания для выполения**

**1.** (0;1) – 3, (0;2) – 9, (1;2) – 5, (2;4) – 1, (1;3) – 8, (2;3) – 2, (3;5) – 4, (4;5) – 6.

**2.** (0;1) – 4, (0;2) – 5, (1;2) – 8, (2;4) – 3, (1;3) – 11, (2;3) – 5, (3;5) – 3, (4;5) – 6.

**3.** (0;1) – 3, (0;2) – 9, (1;2) – 12, (2;4) – 1, (1;3) – 2, (2;3) – 3, (3;5) – 10, (4;5) – 5.

**4.** (0;1) – 6, (0;2) – 2, (2;1) – 3, (2;4) – 6, (1;3) – 1, (2;3) – 5, (3;5) – 8, (4;5) – 7.

**5.** (0;1) – 6, (0;2) – 5, (1;2) – 1, (2;4) – 6, (1;3) – 7, (2;3) – 6, (3;5) – 8, (4;5) – 7.

**6.** (0;1) – 3, (0;2) – 2, (2;1) – 1, (2;5) – 3, (1;5) – 4, (5;4) – 8, (5;3) – 5, (3;4) – 3, (4;6) – 2, (3;6) – 4.

**7.** (0;1) – 10, (0;2) – 5, (2;1) – 4, (2;5) – 8, (1;5) – 3, (5;4) – 4, (5;3) – 2, (3;4) – 1, (4;6) – 5,

(3;6) – 7.

**8.** (0;1) – 3, (0;2) – 2, (2;1) – 2, (2;5) – 12, (1;5) – 8, (5;4) – 2, (5;3) – 6, (3;4) – 1, (4;6) – 8,

(3;6) – 3.

**9.** (0;1) – 2, (0;2) – 7, (2;1) – 1, (2;5) – 6, (1;5) – 12, (5;4) – 10, (5;3) – 5, (3;4) – 4, (4;6) – 2,

(3;6) – 7.

**10.** (0;1) – 4, (0;2) – 2, (2;1) – 1, (2;5) – 7, (1;5) – 5, (5;4) – 4, (5;3) – 1, (3;4) – 4, (4;6) – 3,

(3;6) – 7.

**11.** (0;2) – 2, (0;1) – 7, (2;1) – 4, (2;4) – 9, (1;3) – 3, (3;4) – 1, (4;6) – 2, (3;5) – 8, (6;5) – 4,

(6;7) – 10, (5;7) – 5.

**12.** (0;2) – 10, (0;1) – 5, (2;1) – 1, (2;4) – 4, (1;3) – 3, (3;4) – 5, (4;6) – 3, (3;5) – 10, (6;5) – 10,

(6;7) – 5, (5;7) – 1.

**13.** (0;2) – 4, (0;1) – 6, (2;1) – 4, (2;4) – 6, (1;3) – 3, (3;4) – 2, (4;6) – 4, (3;5) – 7, (6;5) – 3,

(6;7) – 8, (5;7) – 5.

**14.** (0;2) – 3, (0;1) – 1, (2;1) – 4, (2;4) – 2, (1;3) – 6, (3;4) – 4, (4;6) – 3, (3;5) – 6, (6;5) – 4,

(6;7) – 12, (5;7) – 7.

**15.** (0;2) – 8, (0;1) – 12, (2;1) – 3, (2;4) – 6, (1;3) – 5, (3;4) – 4, (4;6) – 10, (3;5) – 4, (6;5) – 6,

(6;7) – 10, (5;7) – 6.

**16.** (0;1) – 3, (0;2) – 3, (1;4) – 3, (2;5) – 3, (1;3) – 2, (0;3) – 4, (2;3) – 2, (3;4) – 2, (3;6) – 4,

(3;5) – 2, (4;6) – 3, (5;6) – 3.

**17.** (0;1) – 3, (0;2) – 2, (1;4) – 13, (2;5) – 13, (1;3) – 7, (0;3) – 11, (2;3) – 9, (3;4) – 5, (3;6) – 10,

(3;5) – 3, (4;6) – 4, (5;6) – 7.

**18.** (0;1) – 4, (0;2) – 5, (1;4) – 7, (2;5) – 7, (1;3) – 5, (0;3) – 8, (2;3) – 4, (3;4) – 2, (3;6) – 7,

(3;5) – 1, (4;6) – 4, (5;6) – 6.

**19.** (0;1) – 5, (0;2) – 5, (1;4) – 4, (2;5) – 5, (1;3) – 3, (0;3) – 10, (2;3) – 3, (3;4) – 3, (3; 6) – 10,

(3;5) – 3, (4;6) – 7, (5;6) – 5.

**20.** (0;1) – 6, (0;2) – 7, (1;4) – 18, (2;5) – 19, (1;3) – 8, (0;3) – 14, (2;3) – 6, (3;4) – 8, (3;6) – 14,

(3;5) – 5, (4;6) – 5, (5;6) – 9.

**21.** (0;1) – 2, (1;2) – 3, (0;2) – 6, (2;3) – 4, (3;4) – 2, (2;4) – 7, (4;5) – 5, (5;6) – 3, (4;6) – 9.

**22.** (0;1) – 3, (1;2) – 5, (0;2) – 7, (2;3) – 6, (3;4) – 5, (2;4) – 12, (4;5) – 3, (5;6) – 2, (4;6) – 4.

**23.** (0;1) – 5, (1;2) – 4, (0;2) – 10, (2;3) – 4, (3;4) – 5, (2;4) – 8, (4;5) – 7, (5;6) – 6, (4;6) – 14.

**24.** (0;1) – 2, (1;2) – 4, (0;2) – 5, (2;3) – 4, (3;4) – 5, (2;4) – 8, (4;5) – 3, (5;6) – 2, (4;6) – 4.

**25.** (0;1) – 3, (1;2) – 2, (0;2) – 5, (2;3) – 1, (3;4) – 1, (2;4) – 3, (4;5) – 2, (5;6) – 2, (4;6) – 3.

**26.** (0;1) – 10, (0;2) – 4, (1;4) – 8, (2;5) – 20, (3;1) – 5, (4;3) – 3, (2;3) – 4, (3;5) – 17, (4;6) – 7,

(5;6) – 5.

**27.** (0;1) – 6, (0;2) – 2, (1;4) – 4, (2;5) – 15, (3;1) – 2, (4;3) – 3, (2;3) – 13, (3;5) – 2, (4;6) – 7,

(5;6) – 1.

**28.** (0;1) – 2, (0;2) – 3, (1;4) – 1, (2;5) – 10, (3;1) – 1, (4;3) – 6, (2;3) – 4, (3;5) – 5, (4;6) – 20,

(5;6) – 6.

**29.** (0;1) – 2, (0;2) – 3, (1;4) – 7, (2;5) – 6, (3;1) – 1, (4;3) – 2, (2;3) – 2, (3;5) – 3, (4;6) – 8,

(5;6) – 10.

**30.** (0;1) – 3, (0;2) – 7, (1;4) – 8, (2;5) – 3, (3;1) – 1, (4;3) – 4, (2;3) – 2, (3;5) – 2, (4;6) – 7,

(5;6) – 6.

Содержание отчета о лабораторной работе

1. Титульный лист
2. Цель работы и задание для выполнения
3. Рисунок графа.
4. Результаты расчета минимального пути.
5. Выводы по лабораторной работе.

Контрольные вопросы

1. Что такое путь на графе?
2. Каковы начальные веса вершин в алгоритме Дейкстры?
3. Как вычисляется вес вершины в алгоритме Дейкстры?
4. Каковы основные этапы в алгоритме Дейкстры?
5. Какова сложность алгоритма Дейкстры?
6. Каковы основные этапы в алгоритме Флойда-Уоршелла?
7. Какова сложность алгоритма Флойда-Уоршелла?