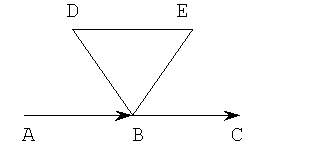
**Решение задачи 23.**

Алгоритм нахождения второго минимального расстояния между двумя вершинами:

По алгоритму Дейкстры находим кратчайший путь между начальной вершиной N и конечной K, состоящий из дуг a1, ..., as. Во втором по длине кратчайшем пути из N в K не будет по крайней мере одного из ребер ai. Поэтому алгоритм нахождения этого второго пути будет следующим:

Для i от 1 до s повторять  
удалить из графа ребро (дорогу) ai;  
найти кратчайший путь из N в K в новом графе;  
если его длина меньше рекордной минимальной длины, полученной ранее,  
то запомнить текущий путь и его длину как рекорд;  
восстановить в графе ребро (дорогу) ai;  
конец\_для;



Этот алгоритм можно найти в книге Кристофидеса "Теория графов. Алгоритмический подход". Но в нем не рассматривается следующий случай:

Кратчайший путь -- ABC, второй -- ABDEBC.

Для того, чтобы его обработать, достаточно найти кратчайший нетривиальный путь из каждой вершины в нее же. Для этого в алгоритме Дейкстры достаточно на нулевом шаге не выставлять пометку "Просмотрено" на начальной вершине.

**2) СПОСОБ ХРАНЕНИЯ ГРАФОВ В ПАМЯТИ.**

Чаще всего используют два способа хранения:

* Матрица смежности.

Представляет из себя двумерный массив размером NxN, где N — количество вершин графа. В матрице A[i,j]=1 (или больше, если граф взвешенный, то есть каждое ребро имеет свой вес или стоимость), если существует ребро между вершинами I и J (в ориентированном графе — можно ли добраться из вершины I к вершине J, то есть ребро направленно в одну сторону), или A[i,j]=0 в противном случае. Очевидно, что для ориетнированного графа a[i,j]=a[j,i] (поскольку ориентированным графом называют частный случай неориентированного графа, у которого каждому ребру i->j есть противонаправленный j->i)

* Список рёбер.

Представляет собой двухмерный массив размером Mx2, где M — количество рёбер графа. В Каждая строка описывает:  
sp[i,1] — от какой вершины отходит ребро i  
sp[i,2] — к какой вершине приходит ребро i;  
Отдельно можно завести массив P(M), где P[i] будет хранить вес ребра i;

**const** MaxN = 100; *//максимальное количество вершин*

INF = 1000000000; *//"бесконечность", заданная наперёд величина, во много раз бОльшая максимальному весу рёбер.*

**type** Matrix = **array**[1..MaxN, 1..MaxN] **of** **longint**; *//тип матрицы смежности. M[i,j] > 0, если существует ребро, идущее от вершины i к j*

Spisok = **array**[1..MaxN \* MaxN, 2]**of** **longint**; *//тип матрицы, содержащая список рёбер. Каждая строка описывает ребро. (ребро k соединяет вершины с номерами s[k,1] и s[k,2])*

Ves = **array**[1..MaxN \* MaxN]**of** **longint**; *//тип матрицы, содержащее веса рёбер для списка*

В зависимости от условий и контекста задач, необходимо вести какой-то определённый формат хранения. При этом стоит иметь в виду, что одни алгоритмы работают с матрицей смежности, а другие - со списком рёбер. Но структура графа такова, что не составляет труда преобразовать список в таблицу, и наоборот. Поэтому бывает удобно вести сразу две структуры, так как чаще всего в задачах вводятся именно рёбра, а матрицу смежности построить, исходя из данного списка. Как это реализовать, будет рассмотрено подробнее.

**3)ВВОД И ВЫВОД**

По умолчанию в программах будет рассматриваться ввод с клавиатуры. Я не стану пока описывать процедуры чтения из файла, так как в разных задачах структура файла бывает разной. Но стоит лишь немного изменить нижеприложенные процедуры, как будет получена возможность чтения из файла.

**procedure** Input\_Table(**var** A : Matrix; N : **longint**); *//процедура ввода матрицы смежности A(N, N)*

**var** i, j : **longint**;

**begin**

**for** i := 1 **to** N **do**

**begin**

**for** j := 1 **to** N **do** read(A[i, j]);

        readln;

**end**;

**end**;

**procedure** Input\_Spisok(**var** P : Spisok; **var** V : Ves; M : **longint**); *//процедура ввода списка взвешенных рёбер P(M,2), M - кол-во рёбер.*

**var** i : **longint**;

**begin**

**for** i := 1 **to** M **do** readln(P[i, 1], P[i, 2], V[i]);

**end**;

**4) ОСНОВНЫЕ АЛГОРИТМЫ**

* Поиск в ширину (BFS).

Суть алгоритма заключается в том, чтобы перебрать всех преемников начальной вершины (корневого узла), и дальше по цепочке. Такой алгоритм помогает получить компоненту связности, то есть схему, куда можно прийти из какой-то заданной вершины. Применяя этот алгоритм поочерёдно для всех вершин, можно найти кратчайшее расстояние, оптимальный путь между двумя вершинами и так далее, в зависимости от предложенных условий.

**procedure** BFS(A : Matrix; N, V : **integer**); *//обход в ширину (V - корневая вершина)*

**var** i, ps, pe : **integer**;

visited : **array** [1..MaxN] **of** **boolean**; *//массив посещённости вершин*

q : **array** [1..MaxN] **of** **integer**; *//"очередь" вершин*

**begin** *//в качестве списка очередных вершин будем использовать структуру "очередь"*

    ps := 1; *//начало очереди*

    pe := 1; *//конец очереди*

    q[pe] := v; *//в очередь помещается исходная вершина. На каждом шаге в очередь будем заносить всех преемников данной вершины (назовём их 1-го уровня, т.е. до которых можно дойти напрямую). Затем просматриваем в очереди занесённые ранее вершины 1-го уровня и добавляем в конец очереди вершины 2-го уровня и так далее до опустошения очереди.*

    visited[v] := **TRUE**; *//вершина V посещена*

**while** ps <= pe **do** *//пока очередь не пуста*

**begin**

**for** i := 1 **to** n **do** **if** (A[v, i] <> 0) **and** (**not** visited[i]) **then** *//перебираем все связные с V вершины*

**begin**

            inc(pe); *//и добавляем в очередь*

            q[pe] := i;

            visited[i] := **true**; *//отмечаем вершину I пройденной*

**end**;

        inc(ps); *//переходим к следующей вершине в очереди*

        v := q[ps]; *//и делаем её корневой*

**end**;

**end**;

* Поиск в глубину (DFS)

Алгоритм поиска в глубину описывается следующим образом: для каждой непройденной вершины необходимо найти все непройденные смежные вершины и повторить поиск для них. Используется в качестве подпрограммы в алгоритмах поиска одно- и двусвязных компонент, топологической сортировки. Реализуется проще BFS, но затрат на ресурсов больше, так как здесь главную роль играет рекурсия

**procedure** DFS(A : Matrix; N, V : **integer**); *//обход в глубину (V - текущая вершина)*

**var** i : **integer**;

**begin**

    visited[v] := **TRUE**; *//вершина V посещена*

**for** i := 1 **to** N **do** **if** (A[v, i] <> 0) **and** (**not** visited[i]) **then** *//если ребро между I и V существует и вершина I не была посещена ранее*

**begin**

        DFS(A, i); *//проверяем вершину I*

**end**;

**end**;

* Алгоритм Дейкстры

Находит кратчайшее расстояние от одной из вершин графа до всех остальных. Алгоритм работает только для графов без рёбер отрицательного веса (так как на таком цикле бесконечно будет уменьшатся наилучший путь). На каждом шаге цикла мы ищем вершину с минимальным расстоянием и флагом равным нулю. Затем мы отмечаем её пройденной и проверяем все соседние с ней вершины. Если в ней расстояние больше, чем сумма расстояния до текущей вершины и длины ребра, то уменьшаем его. Цикл завершается когда все вершины будут пройдены. Время работы алгоритма оценивается как O(N^2).

D : **array**[1..MaxN] **of** **integer**; *//массив кратчайших расстояний*

**procedure** Deisktr(A : Matrix; N, s : **integer**); *//s - искомая вершина*

**var** i, j, v, min : **longint**;

**begin**

    visited[s] := **TRUE**; *//вершина S посещена*

**for** i := 1 **to** N **do** D[i] := A[s, i]; *//изначальный массив расстояний*

**for** i := 1 **to** n-1 **do** *//на каждом шаге находим минимальное решение и пытаемся его улучшить*

**begin**

        min := inf;

**for** j := 1 **to** N **do** **if** (**not** visited[j]) **and** (D[j] < min) **then**

**begin**

            min := D[j]; *//минимальное расстояние*

            v := j; *//найденная вершина*

**end**;

**for** j := 1 **to** N **do** **if** (D[j] > D[v] + A[v, j]) **and** (D[v] < inf) **and** (A[v, j] < inf) **then** D[j] := D[v] + A[v, j]; *//пытаемся улучшить решение. Если в ней расстояние больше, чем сумма расстояния до текущей вершины и длины ребра, то уменьшаем его.*

        s := v; *//новая текущая вершина*

        visited[v] := **TRUE**; *//и она отмечается посещенной*

**end**;

**end**;

* Алгоритм Флойда

Также находит массив кратчайших расстояний. Но в отличие от алгоритма Дейкстры, он использует динамическое программирование. На каждом шаге цикла k создаётся массив решений, где w[i,j] содержит минимальное расстояние, где используется используются только вершины 1,2..k и сами i и j. На начальном этапе W копирует матрицу смежности. Тогда на каждом k есть два варианта — минимальный путь идёт через вершину k или нет. Теоретически такой метод гораздо легче реализовать (банальный перебор), но использует больше машинных ресурсов, чем Дейкстра (сложность алгоритма оценивается как O(N^3), но зато ищет минимальные пути между всеми парами точек). W : **array**[1..MaxN, 1..MaxN] **of** **longint**; *//таблица кратчайших путей*

**function** Min(a, b : **longint**) : **longint**;

**begin**

**if** a < b **then** min := a **else** min := b;

**end**;

**procedure** floyd(A : Matrix; N : **integer**);

**var** i, j, k : **integer**;

**begin**

**for** i := 1 **to** N **do** **for** j := 1 **to** N **do** W[i, j] := A[i, j]; *//копируем матрицу смежности в таблицу расстояний*

**for** k := 1 **to** N **do** *//перебираем все наборы вершин (1),(1,2),(1,2,3)...(1,2,3..N)*

**for** i := 1 **to** N **do**

**for** j := 1 **to** N **do** W[i,j] := min(W[i, j], W[i, k] + W[k, j]); *//возможно два варианта: кратчайшее расстояние от i до j проходит через вершину k или нет.*

**end**;

* Алгоритм Краскала.

Находит каркас минимального веса, т.е такой подграф исходного графа, который бы был связным, содержал все вершины исходного графа и суммарный вес рёбер был наименьшим. В этой задаче используется список рёбер. Вначале текущее множество рёбер устанавливается пустым. Затем, пока это возможно, проводится следующая операция: из всех рёбер, добавление которых к уже имеющемуся множеству не вызовет появление в нём цикла (т.\*е. зачем добавлять ребро R(i,j) в подграф, который содержит эти вершины, а значит, от одной можно добраться до другой), выбирается ребро минимального веса и добавляется к уже имеющемуся множеству. Когда таких рёбер больше нет, алгоритм завершён. Массив рёбер должен быть заранее отсортирован во весу (можно привести док-во: если сначала рассматривать ребро R1(i,j)>R2(i,j), то он потом будет удалён, так как мы встретили в списке рёбер R2(i,j), который весит меньше R1, а удаление рёбер в алгоритме не предусматривается).

**procedure** kraskal(V : Spisok; P : Ves; K, N : **longint**); *//поиск подграфа наименьшего веса (метод Краскала). V(K) - данный список рёбер, P - их вес, N - количество вершин*

**type** TSet = **set** **of** **byte**;

**var** i, j, k1, k2, b, count : **integer**;

mn : **array**[1..MaxN]**of** TSet; *//массив множеств*

select : **array**[1..MaxN \* MaxN]**of** **boolean**; *//выбрано ребро или нет*

**begin**

**for** i := k **downto** 1 **do** *//сортировка рёбер по возрастанию веса*

**for** j:=1 **to** i-1 **do** **if** pp[j] > p[j + 1] **then**

**begin**

        b := P[j];

        P[j] := P[j+1];

        P[j+1] := b;

        b := V[j, 1];

        V[j, 1] := V[j+1, 1];

        V[j+1, 1] := b;

        b := V[j, 2];

        V[j, 2] := V[j+1, 2];

        V[j+1, 2] := b;

**end**;

**for** i := 1 **to** N **do** mn[i] := [i]; *//создаём N множеств - подграфов. Каждое содержит по одной вершине: [1],[2],[3],[4]...[N]*

    count := N; *//кол-во подграфов. Если удается найти требуемый подграф, то на выходе должен остаться 1 подграф*

    i := 1;

**while** (count > 1) **and** (i <= k) **do** *//пока есть нерассмотенные рёбра и кол-во подграфов больше одного*

**begin**

**for** j := 1 **to** count **do** **if** V[i, 1] **in** mn[j] **then** k1 := j **else** **if** V[i, 2] **in** mn[j] **then** k2 := j; *//перебираем все имеющиеся подграфы. В k1 и k2 запоминаем номера подграфов, куда входят вершины, которые соединяют ребро I.*

**if** k1 <> k2 **then** *//если это два разных подграфа, т.е. текущее ребро соединяет их*

**begin**

            mn[k1] := mn[k1] + mn[k2]; *//то соедияем подграфы!*

            mn[k2] := [];

            dec(count); *//уменьшаем кол-во подграфов на единицу*

            select[i] := **TRUE**; *//текущее ребро отмечаем как использованное*

**end**;

        inc(i); *//переходим к следующему ребру*

**end**;

**if** count = 1 **then** *//если после процедуры остался один подграф - выводим номера всех использованных рёбер, иначе - условий для существования единственного подграфа нет (хотя существуют задачи, где необходимо вычислить такие рёбра или вершины (смотря от контекста задачи), которые будут соединять найденные подграфы)*

**begin**

**for** i := 1 **to** k **do** **if** select[i] **then** write(i,' ');

**end** **else** write('-1');

**end**;

**end**;

**5) ПРИМЕРЫ РЕШЕНИЯ ЗАДАЧ НА ГРАФЫ**

1) Дано N городов, некоторые из них соединены двусторонними дорогами. Проезда по каждой дороге имеет свою стоимость. Необходимо составить программу, которая по заданной таблице истинности находит путь от города S1 до города S2, суммарная стоимость которая будет минимальна.  
Формат входных данных: на вход подаётся файл, содержащий в первой строке N. S1 и S2. (1<=N<=50, 1<=S1<=N, 1<=S2<=N). Затем в следующих N строках идут числа, описывающие очередную строку матрицы смежности.  
Формат выходных данных: на экран вывести города, которые следуют в искомом пути.  
Пример:

5 2 4

0 7 0 8 12

7 0 1 0 0

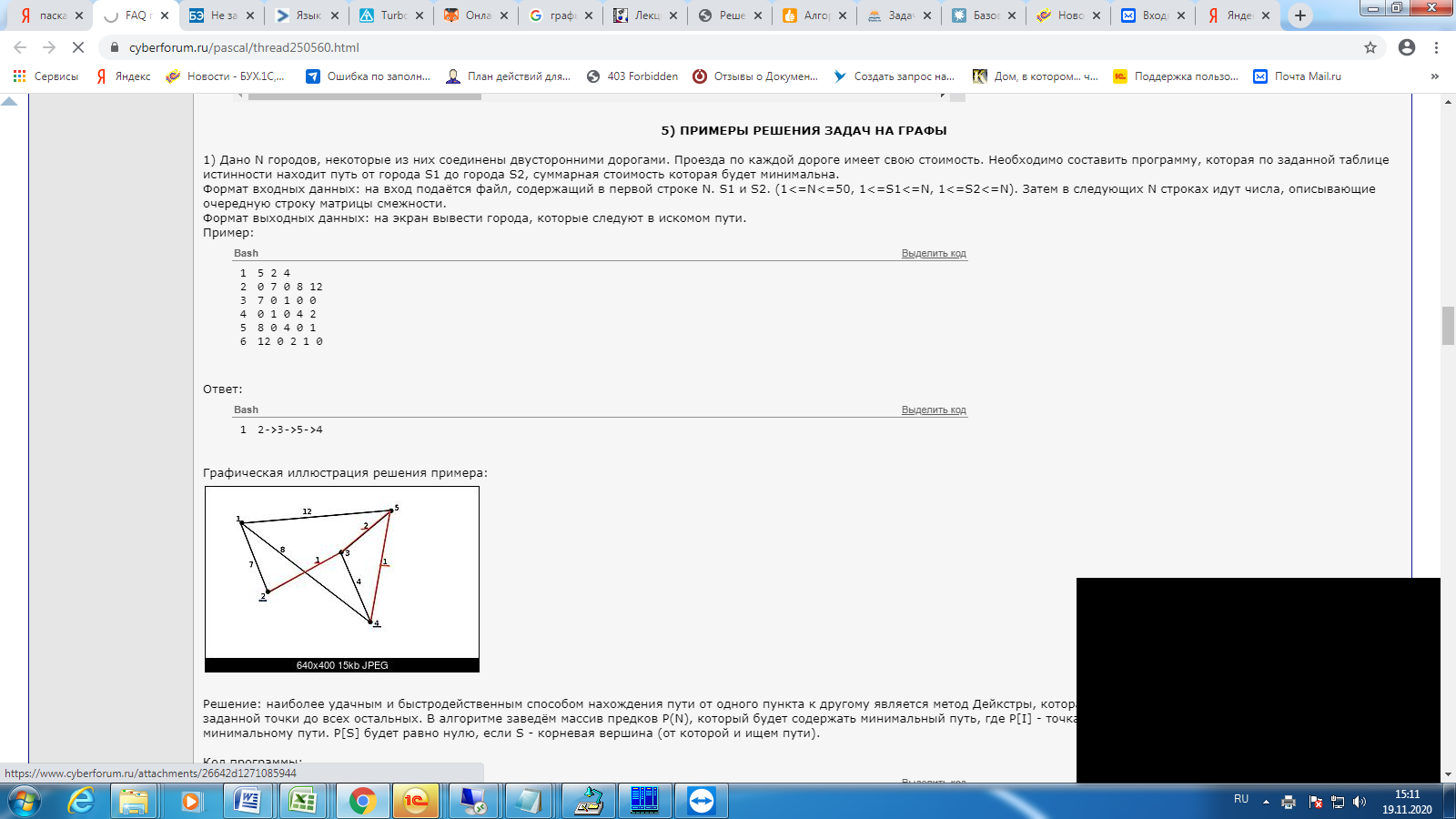
0 1 0 4 2

8 0 4 0 1

12 0 2 1 0

2-**>**3-**>**5-**>**4

Гафическая иллюстрация решения примера:



Решение: наиболее удачным и быстродейственным способом нахождения пути от одного пункта к другому является метод Дейкстры, которая ищет минимальные расстояния от какой-то заданной точки до всех остальных. В алгоритме заведём массив предков P(N), который будет содержать минимальный путь, где P[I] - точка, от которой пришли в I по найденному минимальному пути. P[S] будет равно нулю, если S - корневая вершина (от которой и ищем пути)

**const** MaxN = 50;

INF = 1000000000; *//"бесконечность"*

**type** Matrix = **array**[1..MaxN,1..MaxN] **of** **longint**; *//тип матрицы смежности. M[i,j] = true, если существует ребро, идущее от вершины i к j*

**var**

A : Matrix; N, S1, S2: **integer**;

input: text;

**procedure** Input\_Table(**var** A : Matrix; N : **longint**; **var** T : Text); *//процедура ввода матрицы смежности A(N, N) из текстового файла T*

**var** i, j : **longint**;

**begin**

**for** i := 1 **to** N **do**

**begin**

**for** j := 1 **to** N **do**

**begin**

            read(T, A[i, j]);

**if** (a[i,j] = 0) **and** (i <> j) **then** a[i,j] := INF; *//вершины, которые не связаны ребром, будем обзначать "бесконечностью" ввиду ограничения на вес рёбер*

**end**;

        readln(T);

**end**;

**end**;

**procedure** Deikstr(s, s1 : **integer**); *//s, s1 - искомые вершины (необходимо найти путь от s до s1)*

**var** i, j, v, min, z : **longint**;

st, c : **string**;

visited : **array**[1..MaxN]**of** **boolean**; *//массив посещённости вершин*

D : **array**[1..MaxN] **of** **longint**; *//массив кратчайших расстояний*

P : **array**[1..MaxN] **of** **integer**; *//массив предков, который поможет определить маршрут. p[i] будет содержать предпоследнюю вершину кратчайшего маршрута от s до i*

**begin**

**for** i := 1 **to** N **do**

**begin**

        p[i] := s;

        visited[i] := **FALSE**;

**end**;

    visited[s] := **TRUE**; *//вершина S посещена*

**for** i := 1 **to** N **do** D[i] := A[s, i]; *//изначальный массив расстояний*

    D[s] := 0;

    p[s] := 0; *//*

**for** i := 1 **to** N-1 **do** *//на каждом шаге находим минимальное решение и пытаемся его улучшить*

**begin**

        min := INF;

**for** j := 1 **to** N **do** **if** (**not** visited[j]) **and** (D[j] < min) **then**

**begin**

            min := D[j]; *//минимальное расстояние*

            v := j; *//найденная вершина*

**end**;

**for** j := 1 **to** N **do** **if** (D[j] > D[v] + A[v, j]) **and** (D[v] < INF) **and** (A[v, j] < INF) **then**

**begin**

            D[j] := D[v] + A[v, j]; *//пытаемся улучшить решение. Если в ней расстояние больше, чем сумма расстояния до текущей вершины и длины ребра, то уменьшаем его.*

            p[j] := v;

**end**;

        s := v; *//новая текущая вершина*

        visited[v] := **TRUE**; *//и она отмечается посещенной*

**end**;

    st := ''; *//осталось преобразовать в формат вывода (мы проидёмся по всем вершинам кратчайшего пути от s до s1, но только в обратном порядке)*

    z := p[s1]; *//пока есть корневая вершина*

**while** z <> 0 **do**

**begin**

        str(z,c);

        st := c + '->' + st; *//заносим в маршрут*

        z := p[z]; *//переходим к следующей вершине*

**end**;

    str(s1,c); *//в маршрут записываем начальную вершину*

    st := st + c;

    writeln(st);

**end**;

**BEGIN**

    assign(input,'input.txt');

    reset(input);

    readln(input, N, S1, S2);

    Input\_Table(A, N, input);

    close(input);

    Deikstr(S1, S2);

**END**.

.