**Алгоритм Дейкстри.**

Алгоритм нідерландського вченого Едсгера Дейкстри знаходить всі найкоротші шляхи з однієї наперед заданій вершини графа до всіх інших. З його допомогою, при наявності всієї необхідної інформації, можна, наприклад, дізнатися яку послідовність доріг краще використовувати, щоб дістатися з одного міста до кожного з багатьох інших, або в які країни вигідніше експортувати нафту тощо.

Мінусом даного методу є неможливість обробки графів, в яких є ребра з негативним вагою, тобто, якщо, наприклад, деяка система передбачає збиткові для Вашої фірми маршрути, то для роботи з нею варто скористатися відмінним від алгоритму Дейкстри методом.

Для програмної реалізації алгоритму використовуємо два масиви: логічний **visited** - для зберігання інформації про відвідані вершини та числовий **distance** , в який будуть заноситися знайдені найкоротші шляхи. Отже, є граф **G = (V, E)** . Кожна з вершин входять в множину **n** , спочатку відзначена що не відвідана, тобто, елементам масиву **visited** присвоєно значення **false** .

Оскільки найвигідніші шляхи тільки належить знайти, в кожен елемент вектора **distance** записується таке число, яке точно більше будь-якого потенційного шляху (зазвичай це число називають нескінченністю, але в програмі використовують, наприклад максимальне значення конкретного типу даних). В якості вихідного пункту вибирається вершина **s** і їй приписується нульовий шлях: **distance [s] = 0** , тобто, немає ребра з **s** в **s** (метод не передбачає петель).

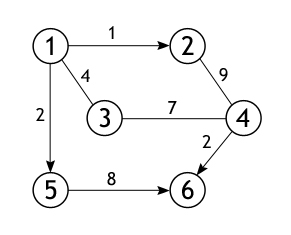
Далі, знаходяться всі сусідні вершини (в які є ребро з **s** ) [нехай такими будуть **t** і **u** ] і по черзі досліджуються, а саме обчислюється вартість маршруту з **s** черзі в кожну з них:

* **distance [t] = distance [s] + вага\_інцідентного s і t ребра;**
* **distance [u] = distance [s] + вага\_інцідентного s і u ребра.**

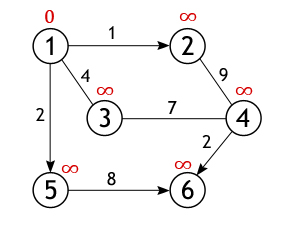
Але цілком ймовірно, що в ту чи іншу вершину з **s** існує кілька шляхів, тому вартість шляху в таку вершину в масиві **distance** доведеться переглядати, тоді найбільше (неоптимальне) значення ігнорується, а найменше ставитися у відповідність вершині.

Після обробки суміжних з **s** вершин вона позначається як відвідана: **visited [s] = true** , і активною стає та вершина, шлях з **s** в яку мінімальний. Припустимо, шлях з **s** в **u** коротший, ніж з **s** в **t** , отже, вершина **u** стане активною і вище описаним способом досліджуються її сусіди, за винятком вершини **s** . Далі, **u** позначається як пройдена: **visited [u] = true** , активною стає вершина **t** , і вся процедура повторюється для неї. Алгоритм Дейкстри триває до тих пір, поки всі доступні з **s** вершини не будуть досліджені.

Тепер на конкретному графі простежимо роботу алгоритму, знайдемо всі найкоротші шляхи між початковою та всіма іншими вершинами. Розмір (кількість ребер) зображеного нижче графа дорівнює **7 (| E | = 7)** , а порядок (кількість вершин) - **6 (| n | = 6)** . Це зважений граф, кожному з його ребер поставлено у відповідність деяке числове значення, тому цінність маршруту необов'язково визначається числом ребер, що лежать між парою вершин.



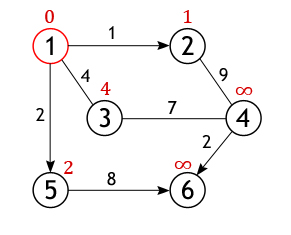
З усіх вершин входять в множину **n** виберемо одну, ту, від якої необхідно знайти найкоротші шляхи до інших доступних вершин. Нехай це буде вершина **1** . Довжина шляху до всіх вершин, крім першої, спочатку дорівнює нескінченності, а до неї - **0** , тобто, граф не має петель.



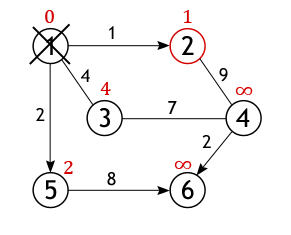
У вершини 1 рівно 3 сусідніх (вершини 2, 3, 5), і щоб обчислити довжину шляху до них потрібно додати вагу дуг, що лежать між вершинами: 1 і 2, 1 і 3, 1 і 5 до значення першої вершини (з нулем) :

**2 ← 1 + 0  
3 ← 4 + 0  
5 ← 2 + 0**

Як уже зазначалося, отримані значення присвоюються вершинам, лише в тому випадку якщо вони «кращі» (мають менше значення) ніж інші на даний момент. А так як кожне з трьох чисел менше нескінченності, вони стають новими величинами, що визначають довжину шляху з вершини 1 до вершин 2, 3 і 5.



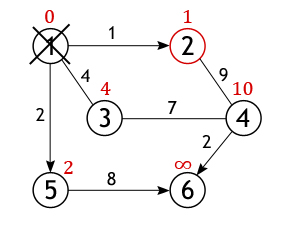
Далі, активна вершина позначається як відвідана, статус «активної» (червоне коло) переходить до однієї з її сусідок, а саме до вершини 2, оскільки вона найближча до раніше активної вершині.



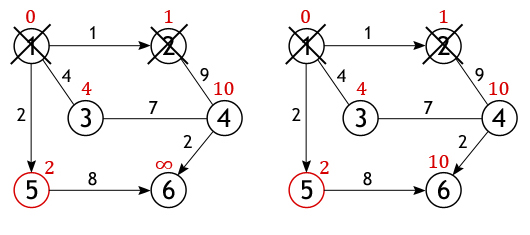
У вершини 2 всього один не розглянутий сусід (вершина 1 позначена як відвідана), відстань до якого з неї дорівнює 9, але нам необхідно обчислити довжину шляху з початкової вершини, для чого потрібно додати величину розраховану для вершини 2 до ваги дуги з неї до вершини 4

**4 ← 1 + 9**

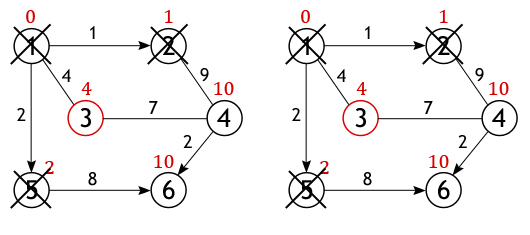
Умова «стислості» (10 <∞) виконується, отже, вершина 4 отримує нове значення довжини шляху.



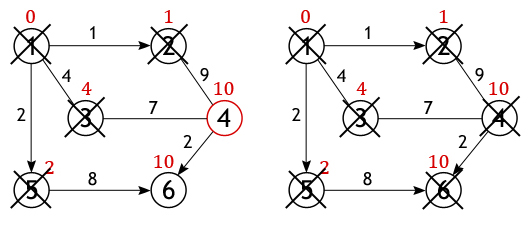
Вершина 2 перестає бути активною, також як і вершина 1 видаляється зі списку не відвіданих. Тепер тим самим способом досліджуються сусіди вершини 5, і обчислюються відстані до них.



Коли справа доходить до огляду сусідів вершини 3, то тут важливо не помилитися, так як вершина 4 вже була досліджена і довжина одного з можливих шляхів з початкової точки до неї обчислено. Якщо рухатися в неї через вершину 3, то шлях становитиме 4 + 7 = 11, а 11 > 10, тому нове значення ігнорується, старе залишається.



Аналогічна ситуація з вершиною 6. Значення найближчого шляху до неї з вершини 1 дорівнює 10, а воно виходить тільки в тому випадку, якщо йти через вершину 5.



Коли всі вершини графа, або всі ті, що доступні з початкової точки, будуть позначені як відвідані, тоді робота алгоритму Дейкстри завершиться, і всі знайдені шляхи будуть найкоротшими. Так, наприклад, буде виглядати список найоптимальніших відстаней шляхів, що лежать між вершиною 1 і всіма іншими вершинами, розглянутого графа:

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

У програмі, що знаходить найближчі шляху між вершинами за допомогою методу Дейкстри, граф буде представлений в вигляді не бінарної матриці суміжності. Замість одиниць в ній будуть виставлені ваги ребер, функція нулів залишиться колишньою: показувати, між якими вершинами немає ребер або ж вони є, але негативно спрямовані.

Джерело: <https://kvodo.ru/dijkstra-algorithm.html>

**Код програми мовою Pascal:**

program Dijkstra;

const

    inf=maxint;

    maxn = 10000;

type

    vektor=array[1..maxn] of longint;

var

    start: longint;

    n: longint;

    GR: array[1..maxn, 1..maxn] of longint;

    distance: vektor;

{algorithm Dijkstra}

procedure Dijkstra;

var

    count, index, i, u, min: longint;

    visited: array[1..maxn] of boolean;

begin

     for i:=1 to n do begin

          distance[i]:=inf;

          visited[i]:=false;

     end;

     distance[start]:=0;

     for count:=1 to n-1 do begin

          min:=inf;

          for i:=1 to n do

              if (not visited[i]) and

                  (distance[i]<=min) then begin

                      min:=distance[i];

                      index:=i;

              end;

          u:=index;

          visited[u]:=true;

          for i:=1 to n do

              if (not visited[i]) and

                 (GR[u, i]<>0) and

                 (distance[u]<>inf) and

                 (distance[u]+GR[u, i]<distance[i]) then

                      distance[i]:=distance[u]+GR[u, i];

     end;

end;

procedure ReadData;

var

    i: longint;

    m,v1,v2,weight: longint;

    f\_in: text;

begin

     Assign(f\_in,'input.txt');

     reset(f\_in);

     readln(f\_in, n, m, start);

     for i:=1 to m do begin

         readln(f\_in, v1, v2, weight);

         GR[v1,v2]:=weight;

         GR[v2,v1]:=weight;

     end;

     close(f\_in);

end;

procedure WriteData;

var

    i: longint;

    f\_out: text;

begin

     Assign(f\_out,'output.txt');

     rewrite(f\_out);

     for i:=1 to n do

         if distance[i]<>inf then

             writeln(f\_out, start,' > ', i,' = ', distance[i])

         else

             writeln(f\_out, start,' > ', i, ' = ', 'no');

     writeln(f\_out);

     close(f\_out);

end;

{main block}

begin

     ReadData;

     Dijkstra;

     WriteData;

end.

**Код програми мовою Python:**

def Dijkstra(n, start, GR):

    inf = 9999999999999999999

    distance = [inf]

    visited = [False]

    for i in range(1, n+1):

        distance.append(inf)

        visited.append(False)

    distance[start] = 0

    for count in range(1,n):

        min = inf

        for i in range(1, n+1):

            if (not visited[i]) and distance[i] <= min:

                min = distance[i]

                index = i

        u = index

        visited[u] = True

        for i in range(1, n+1):

            if (not visited[i]) and (GR[u][i] != 0) and (distance[u] != inf) and (distance[u]+GR[u][i] < distance[i]):

                distance[i] = distance[u]+GR[u][i]

    return n, start, distance

def ReadData():

    f = open('input.txt', 'r')

    l = [line.strip() for line in f]

    n, m, start = map(int, l[0].split())

    GR = [[0]\*(n+1)]\*(n+1)

    # visited = [0]+(n+1)

    for i in range(1, m+1):

        v1, v2, weight = map(int, l[i].split())

        GR[v1][v2] = weight

        GR[v2][v1] = weight

    f.close

    return n, start, GR

def WriteData(n, start, distance):

    inf = 9999999999999999999

    f = open('output.txt', 'w')

    for i in range(1, n+1):

        if distance[i] != inf:

            f.write(str(start) + ' > ' + str(i) + ' = ' + str(distance[i]) +'\n')

        else:

            f.write(str(start) + ' > ' + str(i) + ' = no' + '\n')

    f.write('\n')

    f.close

# main block

n, start, GR = ReadData()

n, start, distance= Dijkstra(n, start, GR)

WriteData(n, start, distance)

**Код програми мовою c++:**

#include <vector>

#include <fstream>

#include <iostream>

const int16\_t MAXN = 100;

int64\_t GR [MAXN][MAXN];

bool visited [MAXN];

int64\_t distance [MAXN];

int64\_t inf = 999999999999;

int64\_t n, start;

void Dijkstra(){

    int64\_t min, index, u;

    distance[0] = inf;

    visited[0] = false;

    for (auto i=1; i<=n; i++){

         distance[i] = inf;

         visited[i] = false;

    }

    distance[start] = 0;

    for (auto count=1; count<=n; count++){

        min = inf;

        for (auto i=1; i<=n; i++){

            if ((not visited[i]) and distance[i] <= min){

                min = distance[i];

                index = i;

            }

        }

        u = index;

        visited[u] = true;

        for (auto i=1; i<=n; i++){

            if ((not visited[i]) and (GR[u][i] != 0) and

                (distance[u] != inf) and

                (distance[u]+GR[u][i] < distance[i])){

                    distance[i] = distance[u]+GR[u][i];

            }

        }

    }

}

void ReadData(){

    using namespace std;

    int64\_t m, v1, v2, weight;

    ifstream fin("input.txt");

    fin >> n >> m >> start;

    for (auto i=1; i<=m; i++){

        fin >> v1 >> v2 >> weight;

        GR[v1][v2] = weight;

        GR[v2][v1] = weight;

    }

    fin.close();

}

void WriteData(){

    std::ofstream fout("output.txt");

    for (auto i=1; i<=n; i++){

        if (distance[i] != inf){

            fout << start << " > " << i << " > " << distance[i] << std::endl;

        } else

        {

            fout << start << " > " << i << " > " << " = no" << std::endl;

        }

    }

    fout.close();

}

int main(){

    ReadData();

    Dijkstra();

    WriteData();

    return 0;

}

**Додатково лекція Тимофея Хирьянова російською мовою.**

<https://www.youtube.com/watch?v=GAarEgus7WQ&t=2885s>

(рос. Мовою)