Пошук найкоротших шляхів на графі.

Пошук найкоротших шляхів на графі є однією з найважливіших задач теорії графів. Для незважених графів очевидно, що найкоротшим буде шлях, що містить найменшу кількість ребер. А тому в найпростішому випадку необхідно просто виконати пошук у ширину і за його допомогою отримати відповідь.

Однак, щоб запам'ятати саме шлях, який привів зі стартової вершини до кінцевої на графі, необхідно трохи модифікувати чергу. Тепер вона буде зберігати не тільки вершини в порядку їх обходу, а й індекс того елементу черги, що містить номер вершини, з якої ми потрапили у дану.

Розглянемо, як буде заповнюватися черга на прикладі графа, що наведений на малюнку 58. Будемо шукати найкоротший шлях між вершинами 1 та 9, причому за стартову вершину візьмемо вершину 9, а за фінішну – вершину 1. Черга, як було відмічено, буде містити по два значення, що описують вершину (можна використовувати запис): перше (поле **ver**) – номер вже відкритої вершини, друге (поле **ind**) – індекс номера вершини у черзі, з якої ми потрапили у поточну.

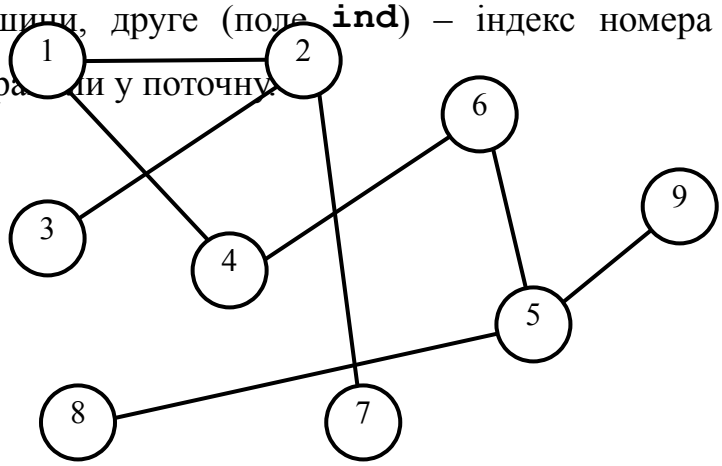


Рис.58

Першою у чергу заносимо фінішну вершину (номер 1 за умовою) і оскільки в неї ми не потрапили з жодної вершини, індекс **ind** дорівнює 0. З вершини 1 пошуком у глибину ми можемо потрапити у вершини 2 та 4, а тому у черзі з'явилося два нових елементи з відповідними індексами (**ind=**1, оскільки саме на такому місці у черзі стоїть батьківська вершина). Далі по черзі буде розглядатися вершина 2. З нею суміжними є вершини 3 та 7, які потрапляють у чергу з індексом **ind**=2 їх «батька». Вершина 4, що стоїть у черзі наступною, «приведе» на обробку тільки вершину 6 (індекс **ind** дорівнює 3), вершини 3 та 7 – не мають суміжних необроблених вершин, тому наступна «продуктивна» вершина – це вершина 6, яка приводить у чергу вершину 5 з індексом «батька» 6, а ця вершина у свою

чергу помістить у чергу дві останні вершини 8 та 9 (індекс «батька» 7). Оскільки стартова вершина потрапила у чергу, процес обходу в ширину можна завершувати. Загалом, після досягнення фінішної вершини черга буде мати наступний вміст:

ver	1	2	4	3	7	6	5	8	9
ind	0	1	1	2	2	4	6	5	5

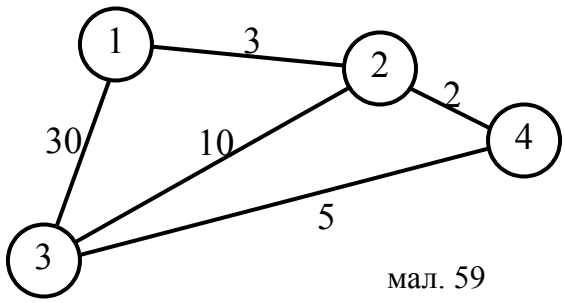
Загалом реалізація алгоритму, крім структури стеку, нічим не відрізняється від пошуку в ширину. Але умовою завершенням алгоритму є не пуста черга, а потрапляння у неї на черговому кроці стартової вершини шляху. Зверніть однак увагу на те що, оскільки невідомо, чи є зв'язним розглядуваний граф, крім перевірки на наявність стартової вершини у «хвості» черги, необхідно ще перевіряти, чи не закінчилася черга. Адже у незв'язному графі задані вершини можуть бути недосяжними.

Для наведеного на рисунку 58 графа відповідь буде мати вигляд (стартова вершина шляху – 9, а фінішна – 1):

```
9-5-6-4-1
```

Довжина знайденого шляху (кількість ребер у ньому) дорівнює 4, що і підраховано у змінній **`len.`**

Для зважених графів таким способом обчислювати шлях не можна, оскільки іноді безпосередній перехід до суміжної вершини може бути менш вигідний, ніж перехід через інші вершини. Приклад такого переходу можна побачити на мал. 59.



мал. 59

Усі можливі переходи з вершини 1 у вершину 3, очевидно, наступні: 1 – 3; 1 – 2 – 3 та 1 – 2 – 4 – 3. Безпосередній перехід (вершини 1 та 3 суміжні) оцінюється 30, перехід через одну вершину 2 – оцінюється 13, а перехід через дві вершини 2 та 4 – оцінюється 10. Найменшим (у теорії графів найкоротшим) є шлях 1 – 2 – 4 – 3.

Існує два алгоритми пошуку мінімальних шляхів на графі (названих на честь їх винахідників) :

- алгоритм Дейкстри – пошук довжин найкоротших шляхів між заданою та всіма іншими вершинами;
- алгоритм Флойда-Уоршалла – пошук довжин найкоротших шляхів між усіма парами вершин графа.

На перший погляд, другий алгоритм є привабливішим, оскільки розв’язує більш загальну задачу, але, на жаль, його складність занадто велика ($\Theta(N^3)$), щоб обробляти великі графи. Для невеликих (порядку 100 вершин) графів він все ж є використовуваним, а тому наведемо обидва.

Алгоритм Дейкстри

Основою цього алгоритму є метод обходу графу пошуком у ширину. Але працює він тільки за умови додатних вагових характеристик ребер. Для ребер з від'ємною вагою його застосувати не можна.

Відомо, що у класичному пошуку в ширину вершини могли мати два кольори: білий (якщо вершина не оброблялася) та сірий (якщо вона оброблялася). В алгоритмі Дейкстри ми розглядатимемо три стани вершин:

- білий – як і раніше, це вершини, які досі не оброблялися;
- чорний – це вершини, обробку яких вже завершено, тобто довжину мінімальної відстані до яких вже знайдено;
- сірий – це вершини, які вже оброблялися, проте для яких ми досі не знаємо, чи знайдено для них довжину справді найменшого шляху.

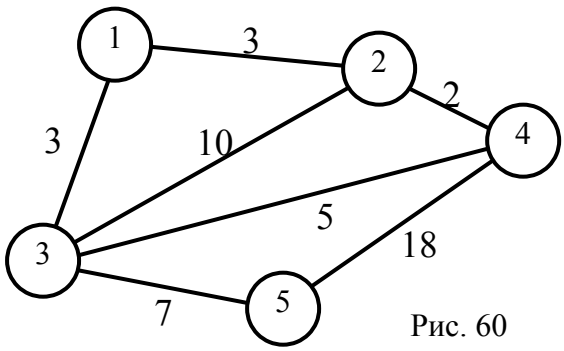


Рис. 60

Для визначеності одразу ж домовимося, що відстань до вершин, недосяжних із заданої, дорівнюватиме машинній нескінченності ∞.

Розглянемо алгоритм на прикладі конкретного графа, зображеного на рис.60. Знайдемо для цього графа довжини найкоротших шляхів від вершини 2 до усіх інших вершин.

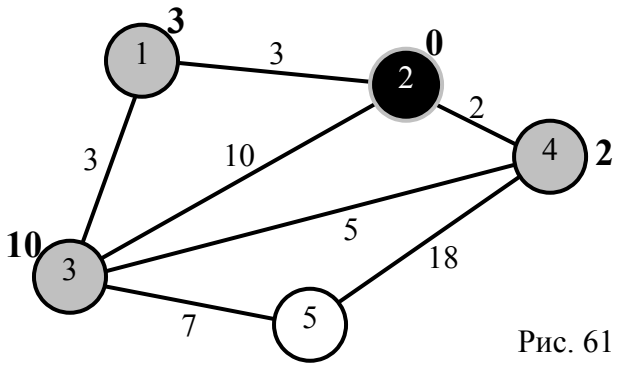


Рис. 61

Позначимо вершину 2 чорним кольором – оскільки ми точно знаємо, що з цієї вершини до неї самої можна дістатися шляхом довжини 0, тобто знаємо відповідь на поставлене питання і обробка цієї вершини завершена.

Із цією вершиною суміжні три вершини: 1, 3 та 4. Їх позначимо сірим кольором. Вочевидь, на даний момент ми точно знаємо, що принаймні шляхом довжини 3 ми можемо дістатися до вершини 1, шляхом довжини 10 – до

вершини 3 та шляхом довжини 2 до вершини 4 (рис.61). Цих результатів ми досягнемо, якщо підемо з вершин 2 по прямих ребрах у ці вершини.

Оберемо серед усіх сірих вершин ту, що на даний момент знаходиться на найменшій відстані від початкової – це вершина 4, до якої можна дістатися шляхом довжини 2. Вочевидь, жоден інший шлях у цю вершину не буде коротшим за знайдений, тому можна позначити вершину 4 чорним кольором.

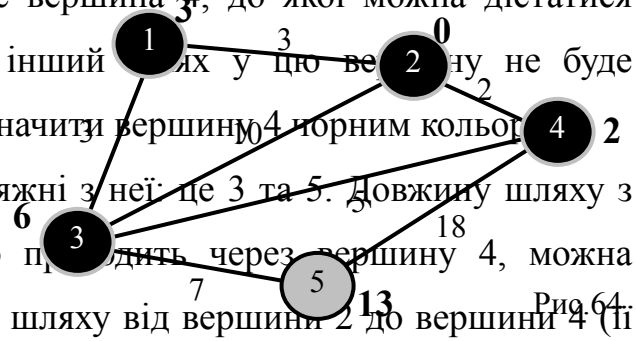


Рис.64

Розглянемо білі та сірі вершини, досяжні з неї: це 3 та 5. Довжину шляху з початкової вершини у вершину 3, що проходить через вершину 4, можна визначити як суму мінімальної довжини шляху від вершини 2 до вершини 4 (її ми вже знайшли – це 2) та довжини ребра з 3 у 4, тобто 5. Отримана довжина 7 є меншою за знайдену раніше 10, тому поновлюємо показник довжини шляху при вершині 3. Вершину 5 позначаємо сірим кольором, а відстань до неї від початкової вершини знову знаходимо як суму 2 та довжини ребра 18, отримуючи 20 (рис. 62).

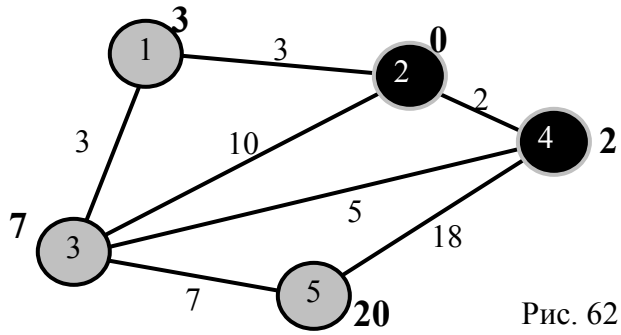


Рис. 62

Знову ж, серед усіх сірих вершин розглянемо найближчу на даний момент до початкової: це вершина 1. Вочевидь, до неї неможливо дістатися коротшим шляхом ніж знайдений, тому позначаємо її чорним кольором. Із нею серед сірих вершин суміжна лише вершина 3. Довжина шляху до вершини 3, який би проходив через вершину 1, одержується як сума довжини мінімального шляху від вершини 2 до вершини 1 (3) та довжини відповідного ребра (3) і дорівнює 6. Це значення менше за знайдене на попередньому кроці (7), тому знову поновлюємо підпис вершини 3 (рис.63).

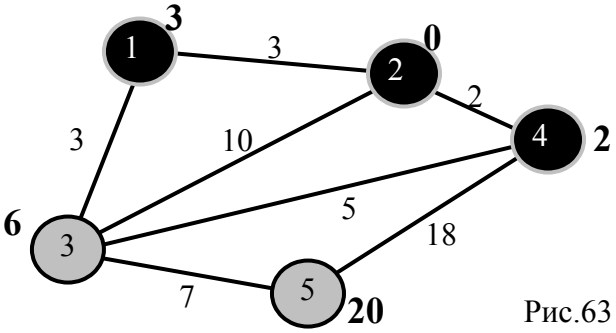


Рис.63

Серед двох вершин, що залишились, вершина 3 має меншу відстань від початкової вершини. До того ж очевидно, що не існує коротшого ніж 6 шляху з

початкової вершини у цю вершину, тому позначаємо її чорним кольором. Сірою лишається тільки вершина 5 і шлях до неї, який проходить через вершину 3, має довжину 6+7=13, що менше за знайдену до поточного моменту відстань (20), тому поновлюємо підпис вершини 5 (рис.64). Зрозуміло, що на цьому обробку завершено і можна перефарбувати вершину 5 у чорний колір.

Варто зазначити, що якщо застосувати описану процедуру до незв'язного графа, наприкінці її роботи «мінімальні» відстані до вершин, що знаходяться поза компонентою зв'язності початкової вершини, будуть невизначеними. То ж якщо на початку роботи покласти відстані до усіх вершин рівними ∞, наприкінці ми одержимо саме той результат, до якого домовилися на початку.

У наведеній нижче частині програми, що реалізує описаний алгоритм, уведено константу **`infinity`** для умовного позначення машинної нескінченності та уведено константи **`white`** та **`black`** для позначення кольорів вершин (сірий колір був використаний нами лише для наглядного пояснення алгоритму, і в явному вигляді перефарбування вершин у сірий колір у реалізації цього алгоритму не буде). Крім того вважається, що матриця суміжності змінюється наступним чином: комірки, що відповідають неіснуючому ребру повинні містити значення нескінченності (на головній діагоналі матриці, однак, залишаються нулі, оскільки довжина шляху від вершини до неї ж самої дорівнює 0). Для реалізації алгоритму використовуються наступні змінні:

- **`matrix`** – матриця суміжності, що представляє граф з **`n`** вершин;
- **`start`** – вершина, з якої починатиме роботу алгоритм;
- **`path`** – результат роботи процедури, тобто лінійний масив найкоротших відстаней від вершини **`start`** до усіх інших вершин графа;
- **`color`** – масив кольорів вершин;
- **`next`** – вершина, у яку буде здійснено перехід на наступному кроці алгоритму (найближча до початкової серед сірих вершин).

Алгоритм завершує свою роботу тоді, коли на черговому його кроці не було знайдено вершини, яку б ще можна було обробити.

```
const Nmax=1000;
```

```
      infinity=MaxLongInt div 2;
      white=0; black=1;
      notfound=0;
type TColors=array[1..NMax]of byte;
     TMatrix=array[1..NMax,1..Nmax]of longint;
     TPath=array[0..Nmax]of longint;
procedure dijkstra(matrix:TMatrix;
    n,start:word;  var path:TPath);
var i,current,next:word;
    color:TColors;
begin
 {Початкові значення: вершини білі,
  мінімальна відстань до них нескінченна}
 for i:=1 to n do
 begin
  path[i]:=infinity;
  color[i]:=white;
 end;
 {Починаємо алгоритм з вершини start}
 path[start]:=0;
 current:=start;
 repeat
  {Поточна вершина стає чорною}
  color[current]:=black;
  next:=notfound;
  for i:=1 to n do
   {перебираємо суміжні з поточною
    білі вершини}
   if (matrix[current,i]<>0)and
      (color[i]=white) then
```

{Зауваження: Оскільки в матриці суміжності там, де немає ребра між двома вершинами, знаходиться

нескінченність, то вказану вище умову можна замінити на наступну:

```
 If matrix[current,i]+path[current]<path[i]
```

Тобто не потрібно перевіряти наявність ребра.}

```
    begin
      {поновлюємо довжину мінімального шляху
       до суміжної вершини}
      if matrix[current,i]+path[current]<path[i]
      then path[i]:=matrix[current,i]+path[current];
      {шукаємо серед «сірих» вершину з
       мінімальною на даний момент відстанню}
      if (next=notfound) or
         (path[i]<path[next])
      then next:=i;
     end;
   current := next;
   {процес завершується, коли не було знайдено
    жодної «сірої» вершини}
  until (current=notfound);
end;
```

Цей алгоритм можна реалізувати і з використанням списків суміжних вершин, який було описано вище. Ідея алгоритму від цього не змінюється, але при пошуку суміжних до поточної вершин початковий індекс ланцюга визначається у масиві **cursor**, а припиняється пошук, якщо у масиві **list** знайдено нульове значення у полі, що визначає номер наступної суміжної вершини.

```
    ...
      color[start]:=black;
      next:=0;
      k:=cursor[start];
      while(k<>0)do
      begin
```

```
      i:=list[k].ver;
      w:=list[k].weight;
      if (color[i]=gray)
      begin
        if len[start]+w<len[i]
        then len[i]:=len[start]+w;
        if len[i]<len[next]
        then next:=i;
      end;
      k:=list[k].next;
    end;
  ...
```

Можна розглядати сірі вершини, як вершини, що знаходяться у пріоритетній черзі, де голова черги містить вершину з найменшою до неї відстанню від стартової. Цей підхід значно прискорює алгоритм, оскільки не вимагає пошуку чергової вершини з мінімальною відстанню від розглядуваної. Але побудова такої черги потребує від програміста певної кваліфікації і тому ми пропонуємо спрощений варіант реалізації алгоритму пошуку найкоротших від заданої вершини шляхів.

У запропонованій реалізації ми з чергою будемо працювати, як зі стандартною (елементи вибираються з голови, а додаються у хвіст черги). Єдине, що буде відрізняти цю реалізацію від стандартного пошуку у ширину, це ознака, за якою вершина потрапляє у чергу: а саме, у чергу будуть додаватися вершини, до яких на заданому кроці було отримано шляхи, коротші, ніж містяться у результуючому масиві шляхів. Крім того, скасовується поняття множини відвіданих вершин, оскільки можливі випадки, коли в одну й ту саму вершину можна потрапити кількома шляхами через інші вершини. Це також призводить до того, що одна й та сама вершина може потрапити до черги кілька разів (причому іноді одночасно) і тому дуже важко порахувати необхідну для резервування довжину черги.

Щоб запобігти одночасному перебуванню однієї вершини кілька разів у черзі рекомендуємо замість поняття відвіданих вершин увести поняття множини вершин, які в даний момент знаходяться в черзі. Ця множина може також мати вигляд масиву булівських змінних, в якому відповідний елемент буде мати значення **True**, якщо вершина в даний момент в черзі, і **False** – у протилежному випадку. Використання такого масиву також забезпечує гарантію того, що одночасно в черзі можуть знаходиться усі вершини (в найгіршому випадку), але тільки по одному разу. Звідси випливає, що довжину черги можна знову зробити рівною кількості вершин в графі, але бажано доступ до неї зробити як до кільцевої структури, тобто при досягненні останнього елемента масиву переходити до першого. Це робиться нескладним способом:

`head:=(head+1)mod N;` - доступ до «головного» елемента черги

або

`tail:=(tail+1) mod N;` - доступ до «хвостового елемента черги

в залежності від того вилучається або додається елемент з черги, але індексацію в цьому випадку зручніше робити з нуля.

Описана реалізація є досить ефективною і простіше сприймається учнями.

Перелічимо тепер остаточно змінні, які будуть використані при реалізації алгоритму:

- **`start`** – номер вершини, з якої запускається пошук найкоротших шляхів;
- **`matrix`** – квадратний масив розмірністю $N \times N$ (**N** – кількість вершин у графі), що є зміненою матрицею суміжності для розглядуваного графа;
- **`len_path`** – результат роботи процедури, тобто лінійний масив найкоротших відстаней від вершини **`start`** до усіх інших вершин графа;
- **`status`** – лінійний масив булівських значень для зберігання станів вершин (знаходиться зараз вершина у черзі чи ні);

- **queue** – лінійний масив розмірністю **N** (оскільки кожна вершина потрапить у неї не більше одного разу) для моделювання роботи черги;
- **next** – вершина, у яку буде здійснено перехід на наступному кроці алгоритму.

Процедура, що реалізує описаний алгоритм мовою Паскаль, буде мати наступний вигляд:

```
const Nmax=1000;
      infinity=MaxLongInt div 2;
type TMatrix=array[1..Nmax,1..Nmax] of longint;
Tpath=array[1..Nmax] of longint;
Procedure Path_short(matrix:TMatrix; N,start:word;
         Var Len_Path:Tpath);
var queue:array[0..Nmax] of word;
head,tail,i,next:word;
status:array[1..Nmax] of boolean;
begin
  {Встановлення початкових значень}
  fillchar(status,sizeof(status),0);
  for i:=1 to N do
    Len_Path[i]:=infinity;
  {Занесення у чергу та список відвіданих вершин
   стартової вершини}
  queue[0]:=start;
  head:=0; tail:=1;
  Len_Path[start]:=0;
  status[start]:=true;
  while head<tail do
  begin
    {Вибір з голови черги першої вершини
     для її розгляду}
```

```
      next:=queue[head];
      head:=(head+1)mod N;
      {Вершина забрана з черги}
      status[next]:=false;
      {Перевірка всіх суміжних з розглядуваною вершин}
      for i:=1 to N do
        if (Len_Path[next]+matrix[next,i]<Len_Path[i])
        then begin
           {До вершини знайдено коротший шлях}
            Len_Path[i]:=Len_Path[next]+matrix[next,i];
            if not status[i]
            then begin
               {Вершина додається в чергу,
                якщо вона там не знаходиться}
               queue[tail]:=i;
               tail:=(tail+1)mod N;
               status[i]:=true;
            end;
        end;
    end;
end;
```

Алгоритм Флойда-Уоршалла

Як вже було сказано, цей алгоритм дозволяє знайти найкоротші шляхи між усіма парами вершин графа. Теоретично можна було б за основу взяти алгоритм Дейкстри та виконати його стільки разів, скільки вершин у графі, вважаючи початковою по черзі кожну з вершин графу. Однак швидкість роботи такого алгоритму, виявляється, буде порівняною зі швидкістю алгоритму Флойда-Уоршала, у той час як реалізація займе значно більше часу.

Тому авторами вищезгаданого алгоритму було запропоновано наступний підхід. Оскільки ми шукаємо довжини шляхів між усіма парами вершин графа, відповіддю буде квадратна таблиця (**matrix**), на перетині *i*-того та *j*-того стовпчика якої буде довжина відповідного шляху. У початковому стані ця таблиця буде співпадати з матрицею суміжності за виключенням того, що несуміжні вершини на перетині стовпчика та рядка будуть мати значення машинної нескінченності, а діагональні елементи будуть мати значення 0 (шлях з вершини в неї саму має нульову довжину). Будемо знаходити по черзі всі шляхи між *i*-тою та *j*-тою вершинами, вважаючи у якості проміжної деяку *k*-ту вершину (у якості проміжних по черзі будемо брати всі вершини графу). Якщо на якомусь кроці ми знайдемо шлях, коротший за вже отриманий, оновимо його у відповідній комірці таблиці (рис.65).

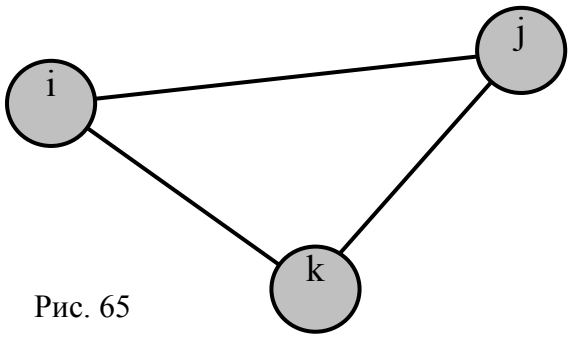


Рис. 65

Програмна реалізація описаного алгоритму очевидна:

```
for k:=1 to n do
  for i:=1 to n do
    for j:=1 to n do
      if matrix[i,j]>matrix[i,k]+matrix[k,j]
      then matrix[i,j]:=matrix[i,k]+matrix[k,j];
```