## МЕТОДИ ОБХОДУ ГРАФІВ

Під ***обходом графа*** розуміють систематичне переміщення по ребрах графа, при якому відвідуються всі вершини графа. Алгоритм обходу графа може багато розповісти про його структуру, а тому деякі інші алгоритми починають свою роботу з отримання інформації про граф шляхом його обходу. Крім того, багато алгоритмів для роботи з графами організовані, як удосконалення базових алгоритмів обходу графа.

Обхід можна виконувати одним з двох способів: *обходом в глибину* та *обходом в ширину*. При обході в глибину прохід по обраному шляху виконується настільки глибоко, наскільки це можливо, а при обході по рівням (пошук у ширину) ми рівномірно рухаємося вздовж усіх можливих напрямків. В обох випадках одна з вершин обирається у якості стартової. Розглянемо детально обидві ці стратегії обходу графа.

### Пошук у ширину

Пошуком у ширину називають один з методів обходу графу, при якому на кожному кроці виконується перехід до всіх суміжних з поточною вершин. Цей метод традиційно називають BFS (*breadth-first search*) — пошук «спочатку в ширину» або «методом хвилі». Дійсно, він нагадує розповсюдження хвилі по поверхні, оскільки виконується поступовий перехід до всіх вершин, що можуть бути досяжними з даної за один крок від початкової (суміжні), за два кроки (суміжні до попередніх), за три і так далі.

Для реалізації описаного алгоритму мовою програмування необхідно крім структури, що описує граф, створити ще кілька допоміжних структур. Граф буде подано матрицею суміжності.

Щоб не повертатися у вершини, які були вже відвідані, будемо запам'ятовувати їх стан у масиві булівських змінних, що набувають значення **false** якщо вершина досі не відвідана, або **true**, якщо вершина була відвідана на якомусь кроці алгоритму. Список відвіданих вершин можна також зберігати у вигляді множини **(set)** чи масиву множин. Це дає змогу зменшити (приблизно у вісім разів) обсяг пам'яті, що витрачається для цієї структури.

Оскільки програма не може виконувати паралельний перегляд суміжних вершин у межах одного кроку алгоритму, ми будемо проглядати їх послідовно. Однак для правильного виконання алгоритму необхідно обробляти вершини в порядку, в якому вони відкривалися. Тобто спочатку всі ті вершини, які досяжні зі стартової за один крок, потім — усі, що досяжні за два кроки, потім — за три і т.д.

Висновок: вершини, у які можна потрапити раніше, і оброблятися повинні раніше, тобто необхідно реалізувати принцип: **first input — first output** (перший прийшов, першим обслуговується). Цей принцип організовується за допомогою структури, що називають *чергою*, і реалізовані вони можуть бути по-різному.

Пропонуємо реалізацію за допомогою статичного лінійного масиву, доступ до якого здійснюється за допомогою двох змінних:

- **head** (голова) — індекс, що вказує на елемент масиву, який ми будемо зчитувати на черговому кроці;
- **tail** (хвіст) — індекс, що вказує на елемент масиву, у який ми будемо записувати на черговому кроці.

Отже, черга (**queue**) має бути подана статичним лінійним масивом, до елементів якого організовано не прямий доступ, а за спеціальним законом.

Операція занесення елемента до неї матиме вигляд:

```
queue[tail]:=значення, що записується у чергу;
inc(tail);
```

А забирання елементу з черги виконуватиметься так:

```
змінна, куди вибирається значення:=queue[head];
inc(head);
```

Очевидно, що якщо черга порожня, то індекси **tail** та **head** співпадають, у протилежному випадку індекс **head** завжди має значення менше, ніж індекс **tail**, оскільки додавання у чергу має передувати вилученню (обслуговуванню) елемента.

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

- **start** — номер вершини, з якої запускається пошук у ширину;
- **matrix** — квадратний масив розмірністю $N$ x $N$ ($N$ — кількість вершин у графі), що є матрицею суміжності для розглядуваного графа;
- **visited** — лінійний масив булівських значень розмірністю $N$ для зберігання станів усіх вершин (відвідана чи ні);
- **queue** — лінійний масив розмірністю $N$ (оскільки кожна вершина потрапить у неї не більше одного разу) для моделювання роботи черги.

Словесний опис алгоритму буде таким:

1. Структуру **visited**, яка відповідає за відвідані вершини обнуляємо, оскільки на початку алгоритму жодна з вершин ще не відвідана.

2. Заносимо стартову вершину у чергу та відмічаємо її як відвідану в масиві **visited**. Значення індексів для черги встановлюємо відповідно: **head:=1** та **tail:=2**, оскільки перший елемент заповнений стартовою вершиною, а другий — є першим вільним.

3. Поки черга не стала порожньою, вибираємо чергову вершину з її голови та перевіряємо всі суміжні з нею вершини. Якщо знайдено суміжні та ще не відвідані вершини, заносимо їх у чергу та відмічаємо у масиві **visited** як відвідані.

Перевірка черги на наявність у ній вершин для обробки здійснюється порівнянням індексів «голови» та «хвоста». Якщо індекс «голови» менше індексу «хвоста», черга ще не порожня і можна продовжувати обробку вершин, що в ній містяться. Оскільки при описаному підході кожна вершина може потрапити у чергу тільки один раз, на певному етапі вершини перестануть додаватися до черги, а оброблятися будуть, і таким чином, черга колись стане порожньою.

Процедура, що реалізує описаний алгоритм мовою Паскаль, матиме вигляд:

```
const Nmax=10000;
type TMatrix=array[1..Nmax,1..Nmax] of byte;
Procedure BFS(matrix:TMatrix; N,start:word);
var queue:array[1..Nmax] of word;
head,tail,i,v:word;
visited:array[1..Nmax] of boolean;
begin
  {Встановлення початкових значень}
  fillchar(visited,sizeof(visited),0);
  {Занесення у чергу та список відвіданих вершин
   стартової вершини}
  queue[1]:=start;
  head:=1; tail:=2;
  visited[start]:=true;
  while head<tail do
  begin
    {Вибір з голови черги першої вершини
     для розгляду}
    v:=queue[head]; inc(head);
   {Перевірка всіх суміжних з розглядуваною вершин}
    for i:=1 to N do
      if (matrix[v,i]<>0)and(not visited[i])
      then begin
             {Вершина суміжна та ще не відвідана}
             queue[tail]:=i;
             inc(tail);
             visited[i]:=true;
           end;
  end;
  {Обробка отриманих результатів}
end;
```

Під обробкою отриманих результатів розуміється, наприклад, перевірка графа на зв'язаність, для чого необхідно перевірити, чи всі вершини потрапили до списку відвіданих вершин.

Формальними параметрами процедури є матриця суміжності, що описує граф, кількість вершин графа та стартова вершина, з якої запускається пошук.

Якщо необхідно тільки перевірити граф на зв'язаність, можна

дещо збільшити швидкодію алгоритму, якщо врахувати той факт, що у зв'язаному графі кожна вершина має потрапити до черги лише один раз. Тоді у випадку, коли «хвіст» (**tail**) черги дорівнюватиме кількості вершин у графі, можна припинити подальшу обробку. У програмі це призведе до використання додаткової умови виходу з циклу обробки черги:

```
while (head<tail) and (tail<N) do ...
```

Якщо у якості структури, що описує граф, використати списки суміжних вершин, алгоритм працюватиме набагато швидше, оскільки розглядатимуться тільки ті вершини, які суміжні з розглядуваною. У цьому випадку цикл **for**, який здійснює пошук суміжних до розглядуваної вершин шляхом перегляду всіх вершин графа у матриці суміжності, буде замінений циклом **while**. Останній переглядатиме послідовно список тільки тих вершин, що суміжні із даною.

```
...
    {Вибирання з голови черги першої вершини
     для розгляду}
    v:=queue[head]; inc(head);
   {Знаходження індексу основного масиву, з якого
   починається список вершин, суміжних до вершини V}
    i:=cursor[v];
{Рухаємося по масиву суміжних вершин, доки не дійдемо
до хвоста списку,який визначається за умовою i=0}
    while i<>0 do
    begin
      {Номер вершини, суміжної до розглядуваної}
      k:=list[i].ver;
      if (not visited[k])
      then begin
             {Вершина суміжна та ще не відвідана}
             queue[tail]:=k;
             inc(tail);
             visited[k]:=true;
           end;
      {Перехід до наступної суміжної вершини}
      i:=list[i].next;
  end;       ...
```

Використовуючи пошук у ширину, можна знайти кількість компонент зв'язаності розглядуваного графа. Для цього тільки необхідно взяти ще одну змінну для підрахунку кількості компонент та завершувати пошук у ширину не тоді, коли черга стала порожньою, а коли в графі більше не залишилося не відвіданих вершин. Якщо ж черга стане порожньою при наявності ще не відвіданих вершин, запустити пошук повторно з іншої ще невідвідуваної вершини, для чого занести цю вершину в чергу і відмітити як відвідану.

Для реалізації описаного алгоритму необхідні крім вже описаних змінних ще дві:

- `count_ver` — кількість вже відвіданих вершин;
- `count_link` — кількість компонент зв'язаності у графі.

Мовою Паскаль алгоритм матиме вигляд:

```
const Nmax=10000;
type TMatrix=array[1..Nmax,1..Nmax] of byte;
Procedure BFS(matrix:TMatrix; N,start:word);
var queue:array[1..Nmax] of word;
head,tail,i,v,count_link,count_ver:word;
visited:array[1..Nmax] of boolean;
begin
  {Встановлення початкових значень}
  fillchar(visited,sizeof(visited),0);
  head:=1; tail:=1; {черга пуста}
  {Кількість компонент зв'язаності
   та розглянутих вершин нульова}
  count_link:=0;
  count_ver:=0;
  while count_ver<N do
  begin
   {Черга порожня, а вершини ще не всі розглянуті}
    if (head=tail)and(count_ver<N)
    then begin
       {Існує ще одна компонента зв'язаності}
       inc(count_link);
       i:=1;
       {Пропускаємо всі відвідані вершини}
       while visited[i] do inc(i);
```

```
      {Першу не відвідану вершину заносимо в чергу}
        queue[tail]:=i;
        inc(tail); inc(count_ver);
        visited[i]:=true;
      end;
      {Вибір з голови черги першої вершини
       для розгляду}
      v:=queue[head]; inc(head);
    {Перевірка всіх суміжних із розглядуваною вершин}
      for i:=1 to N do
        if (matrix[v,i]<>0)and(not visited[i])
        then begin
               {Вершина суміжна та ще не відвідана,
               а тому заноситься у чергу}
               queue[tail]:=i;
               inc(tail); inc(count_ver);
               visited[i]:=true;
             end;
    end;
  {count_link містить кількість компонент зв'язаності}
end;
```

## ПОШУК У ГЛИБИНУ

Пошуком у глибину називають метод обходу графу, при якому на кожному кроці виконується перехід до однієї із суміжних з поточною вершин.

Цей метод називають DFS (*depth-first search*) — пошук «спочатку в глибину», оскільки на кожному кроці вибирається тільки одна із суміжних з поточною вершин та здійснюється перехід саме в цю вершину («заглиблення»). Щоб не повертатися багаторазово у вже відвідані вершини, як і у попередньому випадку, ведеться список (або множина) відвіданих вершин. Якщо на черговому кроці усі вершини, що суміжні з поточною, вже відвідані (так званий «глухий кут»), виконується повернення («відкочування») до попередньої з розглядуваних вершин і звідти шукається ще не відвідана вершина. Відкочування може виконуватися аж до стартової вершини, та алгоритм припиняє свою роботу, коли зі стартової вершини вже нема куди «заглиблюватися», тобто всі доступні вершини відвідані.

Основний результат цього пошуку — знову ж таки визначення, чи є граф зв'язаним (або підрахунок компонент зв'язаності у графі). Проте деякі зміни в алгоритмі дають змогу вирішувати широкий спектр різноманітних задач.

Розглянемо цей різновид пошуку на прикладі графу, поданого на рис. 2.11.

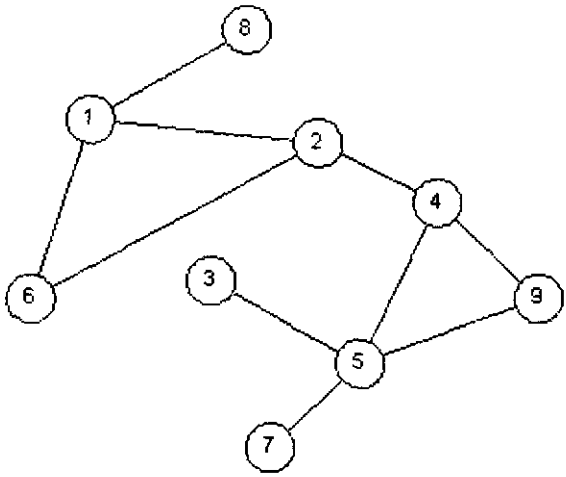


Рис. 2.11

Візьмемо у якості стартової вершину 1. Суміжними з нею є вершини 2, 6 та 8. Візьмемо будь-яку з них, наприклад, 2 та «заглибимося» у неї. Тепер поточною (і вже відвіданою) буде вершина 2, суміжними з якою є вершини 1, 4, 6 та 8. Повертатися в уже відвідану вершину немає сенсу, тому обираємо одну з двох не відвіданих вершин, наприклад, вершину 4.

На основі описаних міркувань ми виконаємо на наступному кроці «заглиблення» у вершину 5, а потім у 3.

Щоразу серед можливих обиралася вершина з найменшим номером, оскільки реалізація передбачається на матриці суміжності (при послідовному проході по рядку цієї матриці ми зупинимося на вершині з найменшим номером, першій серед не відвіданих).

Вершина 3 є «глухим кутом», оскільки не має суміжних не відвіданих вершин, тому буде виконано «повернення» до попередньої відвіданої вершини (5), з якої буде знайдено ще одну не відвідану вершину 7. Ця вершина теж є «глухим кутом», тому знову відбудеться «відкочування» до вершини 5.

З вершини 5 буде знайдено невідвідану вершину 9 (черговий «глухий кут»), з якого після повернення у вершину 5 відбудеться подальше відкочування до вершини 4 (оскільки вершина 5 більше не має суміжних невідвіданих вершин) і далі — у вершину 2.

Із цієї вершини буде знайдено ще одну невідвідану вершину 6, яка, в черговий раз, виявиться «глухим кутом», а тому з неї буде виконано повернення у вершину 2, а потім у вершину 1 (стартову).

Зі стартової вершини буде знайдено останню не відвідану вершину 8, що є «глухим кутом», після якої відбудеться остаточне повернення у стартову вершину 1.

На даному етапі перегляд усіх суміжних зі стартовою вершин не дасть змоги знайти жодної невідвіданої вершини і алгоритм завершить свою роботу.

Таким чином, відвідування вершин наведеного графу при пошуку в глибину, відбуватиметься таким чином:

1—2—4—5—3—5—7—5—9—5—4—2—6—2—1—8—1.

При реалізації такого алгоритму необхідно виконувати спочатку переходи у суміжні невідвідані вершини (заглиблення), а потім повернення назад (відкочування). Причому повернення має здійснюватися чітко у протилежному напрямку, тобто першою з вершин, куди відбудеться відкочування, буде вершина, у яку ми потрапили на передостанньому кроці. У термінах програмування це називається **`last input — first output`** (останнім прийшов — першим обслуговується). Програмно цей процес можна реалізувати рекурсивно або за допомогою структури, що називають *стеком*. Спочатку розглянемо нерекурсивну реалізацію алгоритму обходу з використанням стеку.

***Стек*** — це структура, що працює за принципом **`first input — last output`** (першим прийшов — останнім вийшов) або **`last input — first output`** (останнім прийшов — першим вийшов) і найпростіше реалізується за допомогою лінійного статичного масиву, у який елементи записуються у прямому порядку (від першого до останнього), а зчитуються у зворотному (від останнього до першого).

Візьмемо для моделювання роботи стеку статичний лінійний масив (**stack**), причому для керування необхідним доступом до масиву достатньо однієї змінної **top** (за термінологією вершини стеку). Ця змінна є індексом першого доступного (того, що читається) елементу. При записі ж у масив, ця змінна спочатку збільшується на одиницю, щоб перейти до першого «вільного» елементу, а потім виконується запис.

Занесення елемента у стек, таким чином, матиме вигляд:

```
inc(top);
stack[top]:=значення, що зберігається;
```

а зчитування зі стеку:

```
змінна:=stack[top];
dec(top);
```

У реалізації алгоритму будуть використані такі змінні:

- **start** — номер вершини, з якої запускається пошук у глибину;
- **matrix** — квадратний масив розмірністю $N$ x $N$ ($N$ — кількість вершин у графі), що є матрицею суміжності для розглядуваного графа;
- **visited** — лінійний масив розмірності $N$ булівських значень для зберігання станів усіх вершин (відвідана чи ні);
- **stack** — лінійний масив розмірності $N$ (оскільки кожна вершина потрапить до нього не більше одного разу) для моделювання роботи стеку.

Процедура, що реалізує описаний алгоритм мовою Паскаль, матиме вигляд:

```
const Nmax=10000;
type TMatrix=array[1..Nmax,1..Nmax] of byte;
Procedure DFS(matrix:TMatrix; N,start:word);
var stack:array[1..Nmax] of word;
top,i,v:word;
visited:array[1..Nmax] of boolean;
begin
  {Встановлення початкових значень}
  fillchar(visited,sizeof(visited),0);
  {Занесення у стек та список відвіданих вершин
   стартової вершини}
  stack[1]:=start;
```

```
  top:=1;
  visited[start]:=true;
 {Доки стек не порожній, відбувається заглиблення}
  while top>0 do
  begin
    {Вибирання зі стеку першої вершини
     для розгляду}
    v:=stack[top];
    i:=1;
  {Пошук першої суміжної, але невідвіданої вершини}
    while (i<=N)and((matrix[v,i]=0){не суміжна}
          or(visited[i]){або вже відвідана})
    do inc(i);
    if i<=n
    then begin
       {Знайдено суміжну та невідвідану вершину}
      {Вона додається в стек та список відвіданих
           вершин}
          inc(top);
          stack[top]:=i;
          visited[i]:=true;
        end
    else dec(top);
    {відкочування до попередньої вершини}
  end;
  {Обробка отриманих результатів}
end;
```

Реалізація алгоритму на списку суміжних вершин не викликає труднощів, оскільки ідея алгоритму не змінюється. Зверніть увагу, що після знаходження першої суміжної з поточною, але невідвіданої вершини можна оптимізувати алгоритм, запам'ятавши індекс наступної за знайденою вершини.

Якщо ж знайдена вершина була останньою у списку, відповідний індекс дорівнює нулю, що завадить виконанню циклу **while** при наступному проході.

```
...
  {Вибирання зі стеку першої вершини для її розгляду}
    v:=stack[top];
```

```
{Визначення індексу початку ланцюга суміжних вершин}
   i:=cursor[v];
   {Пошук суміжної, але невідвіданої вершини}
   while (i<>0){список не закінчився або}
     or(visited[list_top[i].ver])
    {вершина відвідана}
   do {перехід до наступної суміжної вершини}
      i:=list_top[i].next;
   if i<>0
   then begin
       {Знайдено суміжну та невідвідану вершину}
         inc(top);
         stack[top]:=list_top[i].ver;
         visited[list_top[i].ver]:=true;
    {Запам'ятовування вершини, на якій зупинилися
          при пошуці суміжних вершин}
         cursor[v]:=list_top[i].next;
       end
   else dec(top);
  {Відкочування до попередньої вершини}
 end;
...
```

Звернемо увагу на те, що описані алгоритми працюють правильно як на незважених, так і на зважених графах, оскільки у матриці суміжності на перетині рядка та стовпчика знаходяться 0, якщо ребро відсутнє (відповідні вершини не суміжні), та ненульове значення у протилежному випадку. Різниця лише у тому, що у зваженому графі це певна кількісна характеристика ребра, а у незваженому лише індикатор його присутності. В принципі можна відноситись до незваженого графу як до графу, в якому вага усіх ребер однакова і дорівнює 1.

Трохи змінивши базовий алгоритм пошуку в глибину, можна знайти шлях між двома заданими вершинами. І хоча цей шлях не обов'язково буде найкоротшим, у деяких випадках його можна використати для розв'язування задачі.

Очевидно, що для реалізації цього пошуку необхідно змінити умову закінчення обходу в глибину. Тепер припиняти роботу алгоритму ми будемо тоді, коли у стек потрапить стартова вершина

шляху (на початку роботи у стек занесемо кінцеву вершину шляху). Враховуючи, що можливий випадок, коли граф є незв'язаним і вершини недосяжні між собою, перевірку стеку на порожність теж залишимо.

Отже, крім стартової вершини (початкової вершини шляху) введемо ще одну змінну — номер кінцевої вершини шляху **fin** і алгоритм мовою Паскаль матиме вигляд:

```
const Nmax=10000;
type TMatrix=array[1..Nmax,1..Nmax] of byte;
Procedure Path(matrix:TMatrix; N,start,fin:word);
var stack:array[1..Nmax] of word;
top,i,v:word;
visited:array[1..Nmax] of boolean;
begin
  {Встановлення початкових значень}
  fillchar(visited,sizeof(visited),0);
  stack[1]:=fin;
  top:=1;
  visited[fin]:=true;
 {Доки стек не порожній, відбувається заглиблення}
  while (top>0) and (stack[top]<>start) do
  begin
    {Вибирання зі стеку першої вершини}
    v:=stack[top];
    i:=1;
 {Пошук першої суміжної, але невідвіданої вершини}
    while (i<=N)and((matrix[v,i]=0){несуміжна}
          or(visited[i]){або вже відвідана})
    do inc(i);
    if i<=n
    then begin
        {Знайдено суміжну та невідвідану вершину}
          inc(top); stack[top]:=i;
          visited[i]:=true;
         end
    else dec(top);
    {відкочування до попередньої вершини}
  end;
  {Якщо стек порожній, вершини недосяжні}
```

```
    {У протилежному випадку стек містить перелік
     вершин, починаючи зі стартової і закінчуючи
     кінцевою, що утворюють між цими вершинами шлях}
end;
```

*При виконанні пошуку в глибину можна проставляти так звані часові мітки.* Кожна вершина має дві такі мітки — першу $d[v]$, в якій вказується, коли (на якому кроці) вершина вперше була відвідана, і другу — $f[v]$, яка фіксує момент, коли пошук закінчує перегляд усіх суміжних з даною вершин, тобто безпосередньо момент завершення обробки вершини перед відкочуванням. Ці мітки являють собою цілі числа в діапазоні від 1 до $2 \cdot |V|$, оскільки для кожної з $V$ вершин існує тільки одна подія її відкриття та одна подія — завершення обробки. Вони використовуються багатьма алгоритмами та корисні при розгляді пошуку в глибину. Очевидно, що для кожної вершини $u$ $d[u] < f[u]$.

Проілюструємо виконання алгоритму пошуку в глибину з підрахунком обох часових міток для графа, що наведений на рисунку 2.12.

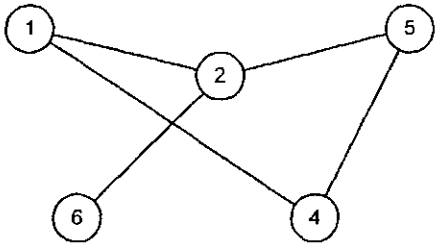


Рис. 2.12

Отриманий результат подано на рис. 2.12, де у вершинах замість їх номерів вказані часові мітки у форматі час відкриття/час завершення.

На рис. 2.13 сірим кольором виділені вершини, в які здійснюється перехід при первинному «заглибленні», а чорним — вже повністю розглянуті вершини, з яких не існує переходів у ще не відвідані суміжні вершини, тобто вершини, з яких виконується остаточний вихід.

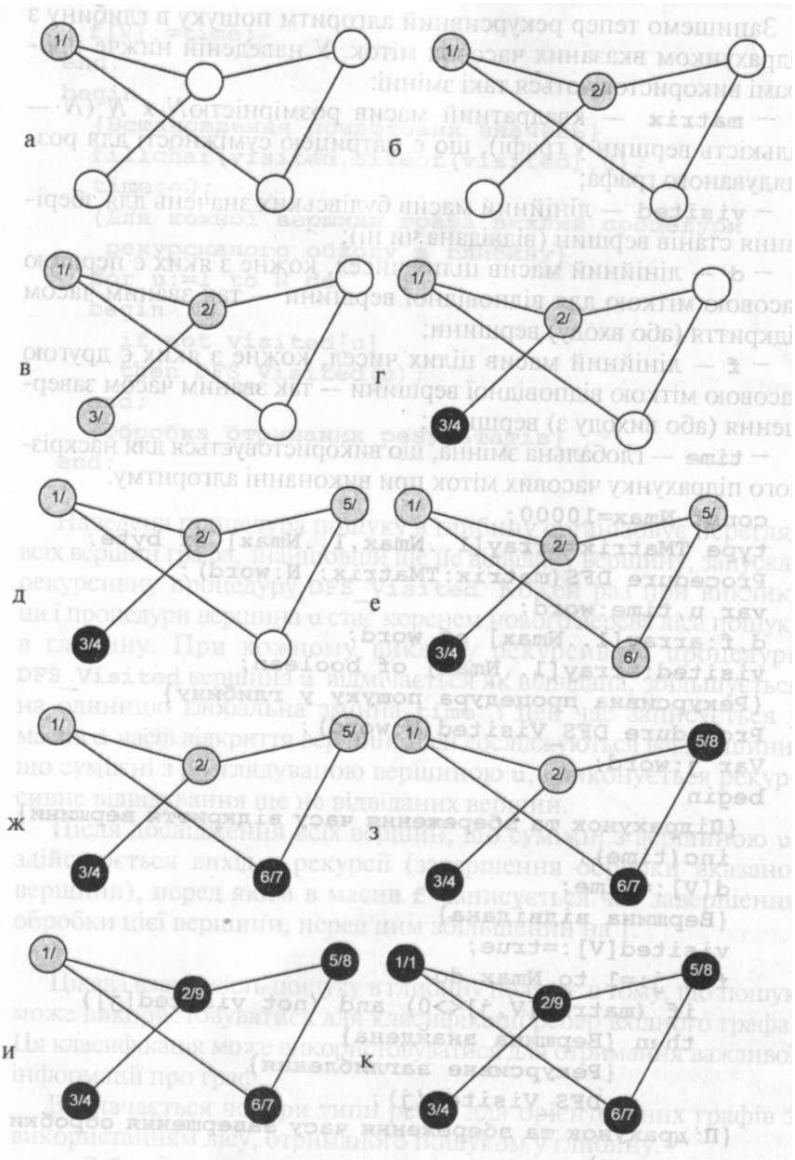


Рис. 2.13. Виконання алгоритму пошуку в глибину

Запишемо тепер рекурсивний алгоритм пошуку в глибину з підрахунком вказаних часових міток. У наведеній нижче програмі використовуються такі змінні:

— **matrix** — квадратний масив розмірністю $N$ x $N$ ($N$ — кількість вершин у графі), що є матрицею суміжності для розглядуваного графа;

— **visited** — лінійний масив булівських значень для зберігання станів вершин (відвідана чи ні);

— **d** — лінійний масив цілих чисел, кожне з яких є першою часовою міткою для відповідної вершини — так званим часом відкриття (або входу) вершини;

— **f** — лінійний масив цілих чисел, кожне з яких є другою часовою міткою відповідної вершини — так званим часом завершення (або виходу з) вершини;

— **time** — глобальна змінна, що використовується для наскрізного підрахунку часових міток при виконанні алгоритму.

```
const Nmax=10000;
type TMatrix=array[1..Nmax,1..Nmax] of byte;
Procedure DFS(matrix:TMatrix; N:word);
var u,time:word;
d,f:array[1..Nmax] of word;
visited:array[1..Nmax] of boolean;
{Рекурсивна процедура пошуку у глибину}
Procedure DFS_Visited(V:word);
Var j:word;
begin
  {Підрахунок та збереження часу відкриття вершини}
  inc(time);
  d[V]:=time;
  {Вершина відвідана}
  visited[V]:=true;
  for j:=1 to Nmax do
    if (matrix[V,j]<>0) and (not visited[j])
    then {Вершина знайдена}
         {Рекурсивне заглиблення}
          DFS_Visited(j);
  {Підрахунок та збереження часу завершення обробки
   вершини}
  inc(time);
```

```
  f[V]:=time;
end;
begin
  {Встановлення початкових значень}
  fillchar(visited,sizeof(visited),0);
  time:=0;
  {Для кожної вершини графа виклик процедури
   рекурсивного обходу в глибину}
  for u:=1 to N do
  begin
    if not visited[u]
    then DFS_Visited(u);
  end;
  {Обробка отриманих результатів}
end;
```

Наведена процедура пошуку в глибину організовує перегляд всіх вершин графа, знайшовши ще не відвідану вершину, запускає рекурсивну процедуру **DFS_Visited**. Кожен раз при виклику цієї процедури вершина **u** стає коренем нового дерева ліса пошуку в глибину. При кожному виклику рекурсивної процедури **DFS_Visited** вершина **u** відмічається як відвідана, збільшується на одиницю глобальна змінна **time** і цей час записується у масив **d** часів відкриття вершин. Далі досліджуються всі вершини, що суміжні з розглядуваною вершиною **u**, і виконується рекурсивне відвідування ще не відвіданих вершин.

Після дослідження всіх вершин, що суміжні з вершиною **u**, здійснюється вихід з рекурсії (завершення обробки вказаної вершини), перед яким в масив **f** записується час завершення обробки цієї вершини, перед цим збільшений на 1.

Цікава властивість пошуку в глибину полягає в тому, що пошук може використовуватися для класифікації ребер вхідного графа. Ця класифікація може використовуватися для отримання важливої інформації про граф.

Визначається чотири типи ребер для орієнтованих графів з використанням лісу, отриманого пошуком у глибину.

1. ***Ребра дерев*** — це такі ребра графа $(u,v)$, для яких при їх дослідженні була відкрита вершина $v$.

2. ***Зворотні ребра*** — це ребра $(u,v)$, що з'єднують вершину $u$ з її предком $v$ у дереві пошуку в глибину. Такими ребрами є ребра-цикли.

3. ***Прямі ребра*** — це ребра $(u,v)$, які не є ребрами дерева та з'єднують вершину $u$ з її нащадком $v$ у дереві пошуку в глибину.

4. ***Перехресні ребра*** — всі інші ребра графа. Вони можуть з'єднувати вершини одного того ж дерева пошуку в глибину, коли жодна вершина не є предком іншої, або з'єднувати вершини різних дерев.

На рис. 2.14 ребра відмічені таким чином: ребра дерев — без мітки, `B` (*back*) — зворотні ребра, `F` (*forward*) — прямі ребра, `C` (*cross*) — перехресні ребра.

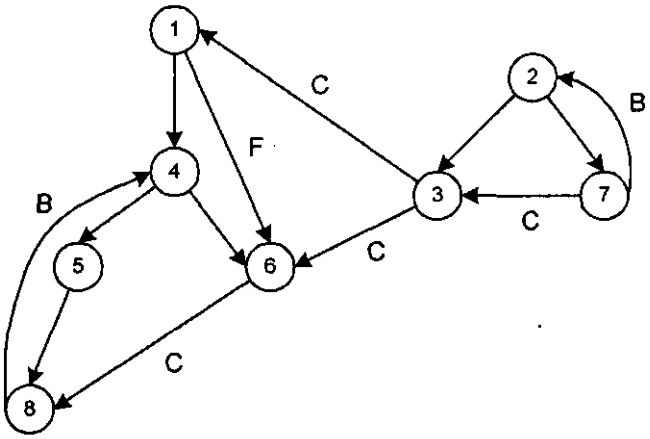


Рис. 2.14. Типи ребер графа

У неорієнтованому графі при класифікації ребер можуть виникнути неоднозначності, оскільки $(u,v)$ та $(v,u)$ насправді є одним ребром.

У такому випадку, якщо ребро відповідає кільком категоріям, воно класифікується відповідно до першої категорії в списку, що застосований до даного ребра.

При пошуку в глибину на неорієнтованому графі будь-яке його ребро є або ребром дерева, або зворотнім ребром.

**Бібліотека «Шкільного світу»**
Заснована у 2003 р.

*Ірина Скляр*

# ТЕОРІЯ ГРАФІВ У ШКОЛІ. ЗАДАЧІ

*Посібник*

**Інформатика. Бібліотека**
**Математика. Бібліотека**

Київ
«Шкільний світ»
2010