Тема 29. Обхід графів

29.1. Складність алгоритмів

Цією лекцією ми починаємо розгляд алгоритмів на графах, тому дамо деякі означення, що стосуються алгоритмів. Під час аналізу алгоритму нас буде цікавити передусім його складність, під якою розуміємо час виконання відповідної програми на комп'ютері. Зрозуміло, що цей показник суттєво залежить від типу та швидкості комп'ютера. Щоб висновки про складність були достатньо універсальними, будемо вважати, що всі обчислення виконуються на якомусь ідеалізованому комп'ютері.

Означення 29.1. **Складність алгоритму** розв'язання задачі – це функція f, яка кожному невід'ємному цілому числу n ставить у відповідність час роботи $f(n)$ алгоритму в найгіршому випадку на входах довжиною n. Час роботи алгоритму вимірюють у кроках (операціях), виконуваних на ідеалізованому комп'ютері.

Наприклад, якщо алгоритм приймає як вхідні дані довільний граф $G = (V, E)$, то під довжиною входу можна розуміти $|V|$ чи $max(|V|, |E|)$.

Розглядаючи вхідні дані достатньо великих розмірів для оцінки такої величини, як порядок зростання часу роботи алгоритму, ми тим самим вивчаємо **асимптотичну ефективність** алгоритмів. Це означає, що нас цікавить тільки те, як час роботи алгоритму зростає зі збільшенням розміру вхідних даних гранично, тобто коли цей розмір зростає до нескінченності. Зазвичай алгоритм, більше ефективний в асимптотичному сенсі, буде більш продуктивним для всіх вхідних даних, за винятком дуже малих.

Для деякої функції $g(n)$ запис $\Theta(g(n))$ (читається «тета від $g(n)$») означає множину функцій:

$$\Theta(g(n)) = \left\{ \begin{array}{c} f(n) : \text{існують додатні константи } c_1, c_2 \text{ та } n_0, \text{такі що} \\ 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ для всіх } n \ge n_0 \end{array} \right\}.$$

Функція $f(n)$ належить множині $\Theta(g(n))$, якщо існують додатні константи c_1 та c_2, які дозволяють заключити цю функцію в межі між функціями $c_1 g(n)$ та $c_2 g(n)$ для достатньо великих n. Оскільки $\Theta(g(n))$ – це множина, то можна записати $f(n) \in \Theta(g(n))$, але також більш поширеним є запис $f(n) = \Theta(g(n))$.

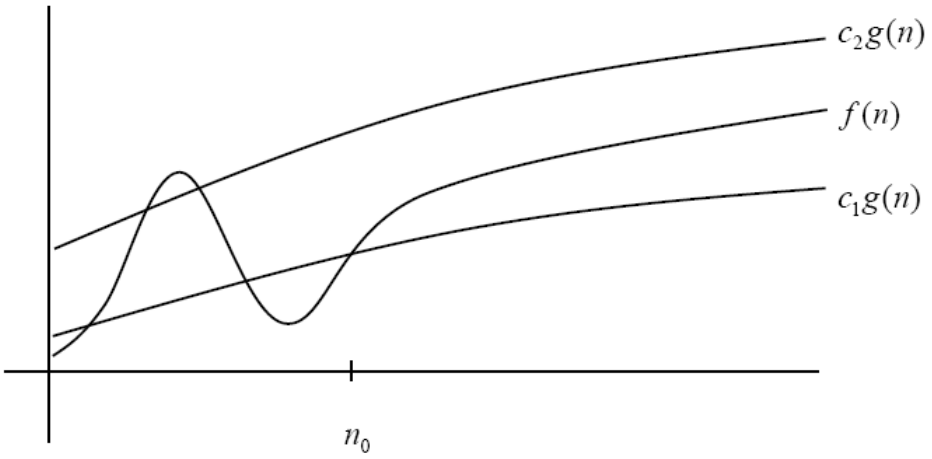


Рис. 29.1. $f(n) = \Theta(g(n))$.

На рис. 29.1 показане інтуїтивне зображення функцій $f(n)$ та $g(n)$, таких що $f(n) = \Theta(g(n))$. Кажуть, що функція $g(n)$ є **асимптотично точною оцінкою** функції $f(n)$.

Відповідно до означення множини $\Theta(g(n))$, необхідно, щоб кожний елемент $f(n) \in \Theta(g(n))$ цієї множини був асимптотично невід'ємним. Це означає, що для достатньо великих n функція $f(n)$ є невід'ємною. Відповідно, функція $g(n)$ повинна бути асимптотично невід'ємною, тому що у зворотному випадку множина $\Theta(g(n))$ виявиться порожньою.

Оскільки будь-яка константа – це поліном нульового ступеню, то сталу функцію можна виразити як $\Theta(n^0)$ або $\Theta(1)$. Але останнє позначення не зовсім точне, адже незрозуміло, по відношенню до якої змінної досліджується асимптотика. $\Theta(1)$ буде часто використовуватись для позначення або константи, або сталої функції деякої змінної.

У Θ-позначеннях функція асимптотично обмежується знизу та згори. Якщо ж достатньо визначити тільки асимптотичну нижню границю, то використовують O-позначення. Для заданої функції $g(n)$ позначення $O(g(n))$ (читається як «омікрон від $g(n)$» або як «велике о від $g(n)$») позначає множину функцій, таких що:

$$O(g(n)) = \{f(n) : \text{існують додатні константи } c \text{ та } n_0, \text{ такі що } 0 \le f(n) \le cg(n) \text{ для всіх } n \ge n_0\}.$$

O-позначення використовується тоді, коли необхідно вказати верхню границю функції з точністю до сталого множника. На рис. 29.2 показане інтуїтивне представлення $f(n) = O(g(n))$.

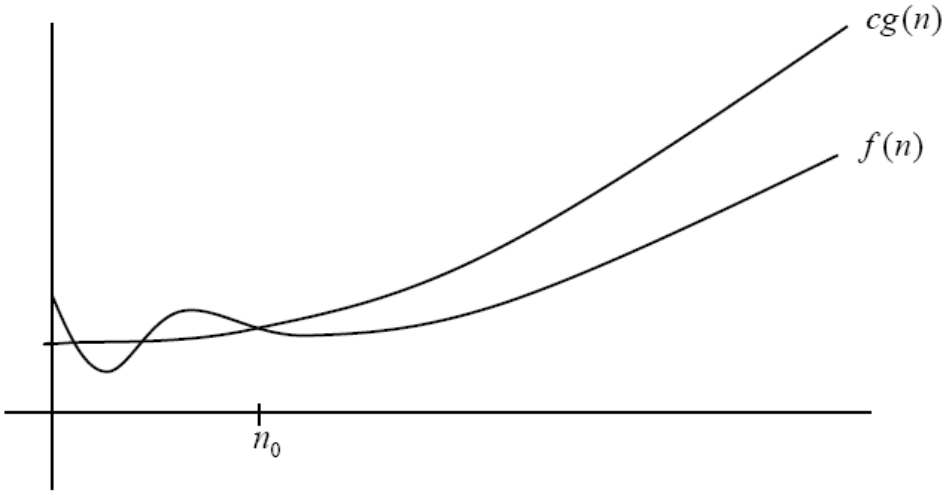


Рис. 29.2. $f(n) = O(g(n))$.

Легко помітити, що якщо $f(n) = \Theta(g(n))$, то $f(n) = O(g(n))$, оскільки Θ-позначення більш сильне, ніж O-позначення (в термінах теорії множин це можна записати як $\Theta(g(n)) \subset O(g(n))$). Таким чином, доведення того, що функція $an^2 + bn + c$, де $a>0$, належить множині $\Theta(n^2)$, одночасно доводить, що множині $O(n^2)$ належить будь-яка подібна квадратична функція.

Аналогічно тому, як в O-позначеннях дається асимптотична верхня границя функції, в Ω-позначеннях дається її асимптотична нижня границя. Для заданої функції $g(n)$ позначення $\Omega(g(n))$ (читається як «омега від $g(n)$») позначає множину функцій, таких що:

$$\Omega(g(n)) = \{f(n) : \text{існують додатні константи } c \text{ та } n_0, \text{ такі що } 0 \le cg(n) \le f(n) \text{ для всіх } n \ge n_0\}.$$

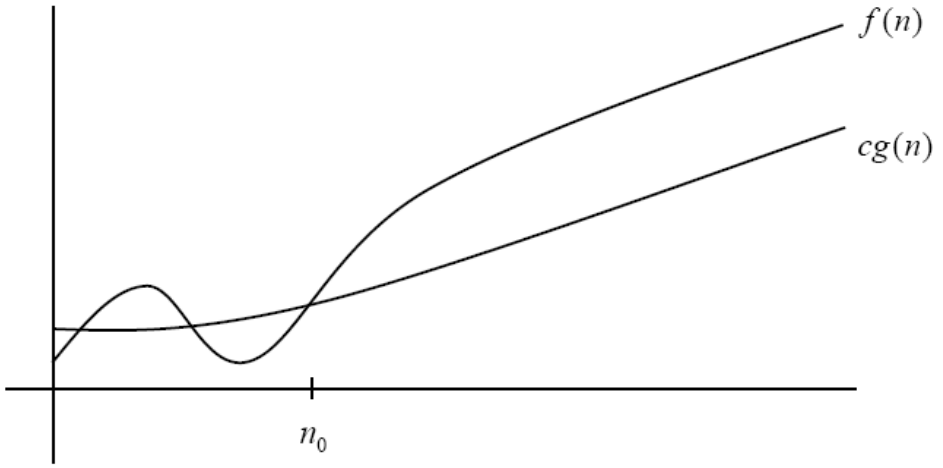


Рис. 29.3. $f(n) = \Omega(g(n))$.

Ω-позначення використовується тоді, коли необхідно вказати нижню границю функції з точністю до сталого множника. На рис. 29.3 показане інтуїтивне представлення $f(n) = \Omega(g(n))$.

Алгоритм зі складністю $O(n)$, де n – довжина входу, називають **лінійним**. Такий алгоритм для переважної більшості задач найліпший (за порядком) щодо складності.

Алгоритм, складність якого дорівнює $O(p(n))$, де $p(n)$ – поліном, називають **поліноміальним**. Часто замість $O(p(n))$ пишуть $O(n^a)$, де a – константа. Всі задачі дискретної математики, які вважають важкими для алгоритмічного розв'язування, нині не мають поліноміальних алгоритмів. Крім того, поняття «поліноміальний алгоритм» – це найпоширеніша формалізація поняття «**ефективний алгоритм**».

Алгоритми, часова складність яких не піддається подібній оцінці, називають **експоненціальними**. Більшість експоненціальних алгоритмів – це просто варіанти повного перебору, а поліноміальні алгоритми здебільшого можна побудувати лише тоді, коли вдається заглибитись у суть задачі. Задачу називають **важкорозв'язною**, якщо для її розв'язання не існує поліноміального алгоритму.

29.3. Обхід графів. Пошук вшир

Існує багато алгоритмів на графах, які ґрунтуються на систематичному переборі їх вершин або обході вершин, під час якого кожна вершина отримує унікальний порядковий номер. Виділяється два основних алгоритми обходів графів або пошуку у графах – це пошук вшир та пошук углиб.

Почнемо з методу **пошуку вшир** або BFS-методу (*breadth first search*). Нехай $G = (V, E)$ – простий зв'язний граф, усі вершини якого позначено попарно різними символами. У процесі пошуку вглиб вершинам графу G надають номери (BFS-номери). У ході роботи алгоритму використовують структуру даних для збереження множин, яку називають чергою. З черги можна вилучити тільки той елемент, який було додано до неї першим: черга працює за принципом «першим прийшов – першим вийшов» (*first in, first out* – скорочено FIFO). Елемент включається у хвіст черги, а виключається з її голови.

Нижче наводиться процедура пошуку вшир BFS. Вона приймає на вхід два параметри: граф G (який може бути представлений будь-яким зручним способом: матрицями суміжності та інцидентності, списком суміжності) та початкова вершина s. Процедура починає обхід графу з вершини s, додаючи у чергу Q нові вершини. Робота закінчується, коли черга Q стає порожньою.

```
BFS(граф G, початкова вершина s)
1.  позначити вершину s як відвідану та приписати їй номер BFS[s] ← 1
2.  внести в чергу Q вершину s: Q ← [s]
3.  k ← 1
4.  while Q ≠ ∅
5.      v ← перша вершина в Q
6.      для кожного ребра (v, u) в G:
7.          if вершина u ще не відвідана (тобто номер BFS[u] не визначений)
8.              k ← k + 1
9.              позначити u як відвідану та приписати їй номер BFS[u] ← k
10.             додати u в кінець черги Q
```

Лістинг 29.1. Процедура BFS пошуку вшир в графі

Щоб результат виконання алгоритму був однозначним, вершини, які суміжні з вершиною v, аналізують за зростанням їх порядкових номерів (або в алфавітному порядку). Динаміку роботи алгоритму зручно відображати за допомогою таблиці з трьома стовпцями: вершина, BFS-номер, уміст черги. Цю таблицю називають протоколом обходу графу пошуком вшир.

Для прикладу виконаємо обхід графу, який зображено на рис. 29.4, починаючи з вершини *b*. Протокол обходу наведено в таблиці 29.1.

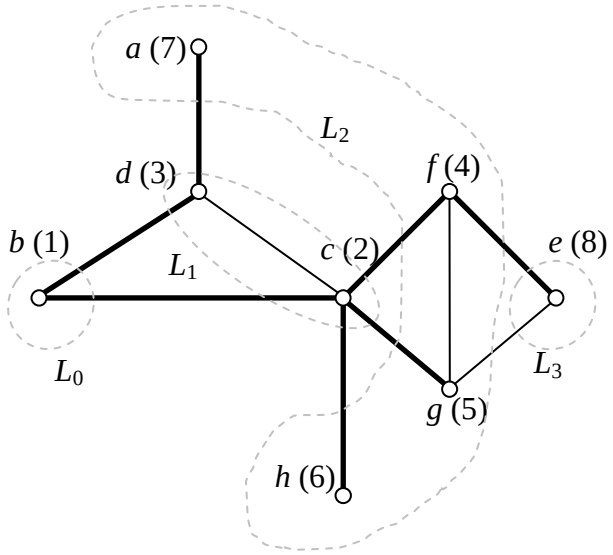


Рис. 29.4.

Вершина	BFS-номер	Вміст черги
b	1	*b*
c	2	*bc*
d	3	*bcd*
-	-	*cd*
f	4	*cdf*
g	5	*cdfg*
h	6	*cdfgh*
-	-	*dfgh*
a	7	*dfgha*
-	-	*fgha*
8	*e*	*fghae*
-	-	*ghae*
-	-	*hae*
-	-	*ae*
-	-	*e*
-	-	∅

Табл. 29.1.

У процесі роботи алгоритму пошуку вшир будується дерево *T* пошуку. Це дерево є підграфом графу *G* і складається з тих ребер (*v*, *u*), які були розглянуті на кроці 6 процедури BFS. Якщо граф *G* зв'язаний, то дерево *T* буде включати в себе всі вершини графу. Таке дерево називається **покриваючим деревом графу**. На рис. 29.4 ребра дерева *T* позначені потовщеними лініями.

Досліджуючи алгоритм пошуку вшир можна прийти до наступних висновків.

Лема 29.1. Якщо вершина *v* графу *G* зустрілась під час роботи процедури BFS(*G*, *s*), то в графі *G* існує шлях від *s* до *v*.

Лема 29.2. Час роботи алгоритму пошуку вшир становить $O(n_s + m_s)$, де n_s – кількість вершин досяжних з *s* в графі *G*, m_s – кількість ребер досяжних з *s* в графі *G*.

Доведення цих лем тривіальне та ґрунтується на аналізі коду процедури BFS (див. лістинг 29.1).

29.3. Найкоротші відстані на основі пошуку вшир

Пошук вшир може використовуватись не тільки для обходу графу, але й для інших практичних задач. По-перше, метод пошуку вшир в графі *G* дозволяє знайти найкоротші шляхи від початкової вершини *s* до будь-якої іншої досяжної від неї вершини *v* графу.

Дійсно, прослідкувавши за роботою алгоритму пошуку вшир, можна помітити, що множина вершин *V* графу *G* в ході роботи алгоритму розбивається на підмножини $L_0, L_1, L_2, \ldots$, де $L_0 = \{s\}$, $L_1 = \{v: (s, v) \in E\}$, $L_2 = \{v: (u, v) \in E$ та $u \in L_1\}$ і т.д., причому $\bigcup_{i=0} L_i = V$ та $L_i \cap L_j = \varnothing$ для всіх $i \neq j$. Ми можемо назвати множини L_i рівнями пошуку вшир. Тоді за визначенням вершини рівню L_i зустрінуться під час обходу графу вшир після того, як будуть розглянуті всі вершини рівня L_{i-1}. На рис. 29.4 вказані рівні позначені сірими контурними лініями.

Вивчаючі рівні L_i пошуку вшир, можна дійти висновку, що всі вершини цього рівня віддалені на відстань *i* від початкової вершини *s*. Причому ця відстань буде мінімальною. Таким чином, можна сформулювати наступну теорему.

Теорема 29.1. Нехай під час роботи процедури BFS(*G*, *s*) пошуку вшир у графі *G* від початкової вершини *s*, всі досяжні від *s* вершини графу утворюють множину рівнів:

$L_0, L_1, L_2, \ldots$, причому $\bigcup_{i=0} L_i = V$ та $L_i \cap L_j = \varnothing$ для всіх $i \neq j$. Тоді відстань від початкової вершини s до будь-якої вершини $v \in L_i$ дорівнює i.

Доведення пропонуємо провести самостійно.

Щоб реалізувати можливість обрахунку найкоротших відстаней від початкової вершини s до всіх інших досяжних від неї вершин, процедура BFS потребує мінімальних змін. Необхідно ввести додатковий параметр $dist[v]$, для будь-якої вершини, ініціалізувавши його 0 для початкової вершини s та $+\infty$ для всіх інших вершин графу. А також після рядка 8 процедури BFS необхідно додати інструкцію: $dist[v] \leftarrow dist[v] + 1$.

29.4. Виявлення компонент зв'язності у неорієнтованих графах

Іншим важливим застосуванням пошуку вшир є знаходження компонент зв'язності (див. тему 26) у неорієнтованих графах. Нагадаємо, що компонентами зв'язності неорієнтованого графу $G = (V, E)$ є класи еквівалентності відношення vRu = «існує маршрут між v та u». Іншими словами, компонента зв'язності – це підграф графу G такий, що в ньому існує маршрут між кожною парою вершин.

Алгоритм пошуку вшир дозволяє легко знайти або підрахувати всі компоненти зв'язаності неорієнтованого графу G. Для цього достатньо запускати процедури BFS, використовуючи в якості початкової такі вершини графу, які ще не були зустрінуті при попередніх викликах. Нижче наводиться псевдокод процедури FindConnectedComponents. Ця процедура повертає кількість знайдених компонент зв'язності. Її також можна легко модифікувати, щоб зберігати самі компоненти, наприклад, у вигляді множин.

```
FindConnectedComponents(граф G)
1. number_of_components ← 0
2. позначити всі вершини графу G не відвіданими
3. for i ← 1 to n
4.       if вершина i ще не відвідана
5.             BFS(G, i)
6.             number_of_components ← number_of_components + 1
7. return number_of_components
```

Лістинг 29.2. Процедура FindConnectedComponents підрахунку кількості компонент зв'язаності неорієнтованого графу

Під час виклику процедури BFS(G, i) в рядку 5 відбувається обхід всіх досяжних від i вершин графу G. Ці вершини разом із i утворять нову компоненту зв'язності. Причому під час даного обходу нові вершини будуть помічені як відвідані і, таким чином, не будуть розглядатись при наступних викликах процедури BFS.

На рис. 29.5 наведений приклад роботи процедури FindConnectedComponents. Наведений тут граф містить три компоненти зв'язності. Тому виклик процедури BFS у рядку 5 відбудеться три рази із початковими вершинами: a, f та h. На рисунку числа біля вершин означаються BFS-номери, які вершини отримують протягом послідовних викликів процедури пошуку вшир.

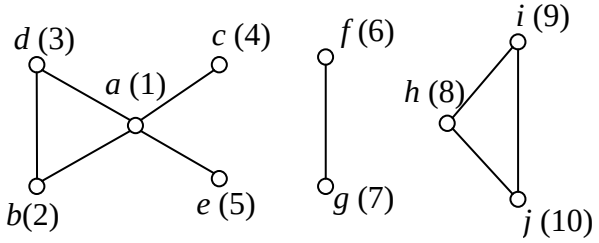


Рис. 29.5. Компоненти зв'язності та результат роботи процедури FindConnectedComponents

Легко показати, що час роботи процедури FindConnectedComponents становить O(n + m), де n і m – відповідно кількість вершин та ребер графу G.

29.5. Пошук углиб

Другий базовий метод обходу графу – алгоритм **пошуку углиб** або DFS-метод (*depth first search*). У процесі його роботи всім вершинам графу надаються номери (DFS-номери). На відміну від методу пошуку вшир, пошук углиб використовує структуру даних стек для збереження вершин-кандидатів. Зі стеку можна вилучити тільки той елемент, який було додано до нього останнім: стек працює за принципом «останнім прийшов – першим вийшов» (*last in, first out* – скорочено LIFO). Інакше кажучи, додавання й вилучення елементів у стеку відбувається з одного кінця, який називається верхівкою стеку.

Таким чином, процедура DFS повністю повторює процедуру BFS, відрізняючись від неї тільки тим, що замість черги тут використовується стек.

```
DFS(граф G, початкова вершина s)
1.  позначити вершину s як відвідану та приписати їй номер DFS[s] ← 1
2.  внести в стек S вершину s: S ← [s]
3.  k ← 1
4.  while S ≠ ∅
5.      v ← вершина в голові стеку S
6.      if існує ребро (v, u) в G таке, що вершина u ще не відвідана
7.          k ← k + 1
8.          позначити u як відвідану та приписати їй номер DFS[u] ← k
9.          додати u в голову стеку S
10.     else
11.         видалити вершину v з голови стеку S
```

Лістинг 29.3. Процедура DFS пошуку вглиб в графі

Аналогічно до пошуку вшир, для відслідковування роботи процедури DFS можна використовувати протокол обходу. Для прикладу виконаємо обхід графу з рис. 29.4, починаючи так само з вершини b. Результат обходу зображено на рис. 29.6, протокол обходу – в таблиці 29.2.

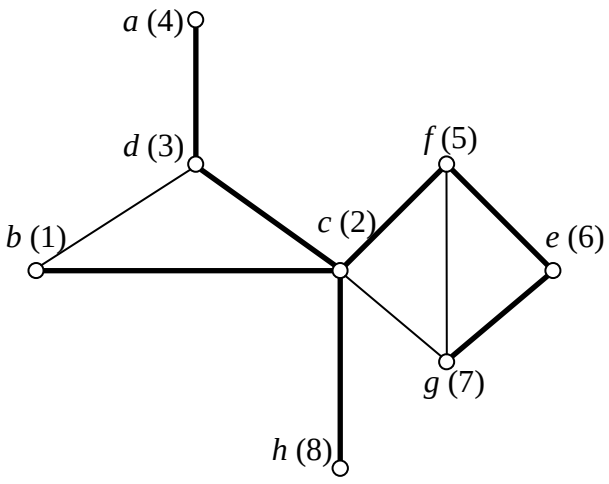


Рис. 29.6.

Вершина	DFS-номер	Вміст стеку
b	1	*b*
c	2	*bc*
d	3	*bcd*
a	4	*bcda*
-	-	*bcd*
-	-	*bc*
f	5	*bcf*
e	6	*bcfe*
g	7	*bcfeg*
-	-	*bcfe*
-	-	*bcf*
-	-	*bc*
h	8	*bch*
-	-	*bc*
-	-	*b*
-	-	∅

Табл. 29.2.

У процесі роботи алгоритму пошуку вглиб, так само як і у випадку пошуку вшир, будується дерево T пошуку. Це дерево є підграфом графу G і складається з тих ребер (v, u), які були розглянуті на кроці 6 процедури DFS. Якщо граф G зв'язаний, то дерево T буде включати в себе всі вершини графу. На рис. 29.6 ребра дерева T позначені потовщеними лініями.

Аналогічно до властивостей пошуку вшир, можна встановити подібні властивості пошуку вглиб в графі.

Лема 29.3. Якщо вершина v графу G зустрілась під час роботи процедури DFS(G, s), то в графі G існує шлях від s до v.

Лема 29.4. Час роботи алгоритму пошуку вглиб становить $O(n_s + m_s)$, де n_s – кількість вершин досяжних з s в графі G, m_s – кількість ребер досяжних з s в графі G.

Окрім наведеного вище псевдокоду для процедури DFS, можна також реалізувати метод обходу графу пошуком вглиб за допомогою рекурсивних викликів. Процедура пошуку вглиб у такому вигляді застосовується у багатьох практичних задачах, деякі з яких описуються в наступних розділах. Дана рекурсивна реалізація у вигляді процедури DFSR наведене нижче.

```
DFSR(граф G, початкова вершина s, поточний DFS-номер k)
1. позначити s як відвідану
2. DFS[s] ← k
3. k ← k + 1
4. для кожного ребра (s, u) в G:
5.         if вершина u ще не відвідана (тобто номер DFS[u] не визначений)
6.                 DFSR(G, u, k)
```

Лістинг 29.4. Рекурсивна процедура DFSR пошуку вглиб в графі

29.6. Топологічне сортування

Пошук углиб в графах має також багато додаткових застосувань. Одним з них є топологічне сортування.

Означення 29.2. Топологічним сортуванням орієнтованого ациклічного графу $G = (V, E)$ називається таке лінійне впорядкування всіх його вершин, що якщо граф містить ребро (v, u), то вершина v в такому впорядкування розташовується до вершини u.

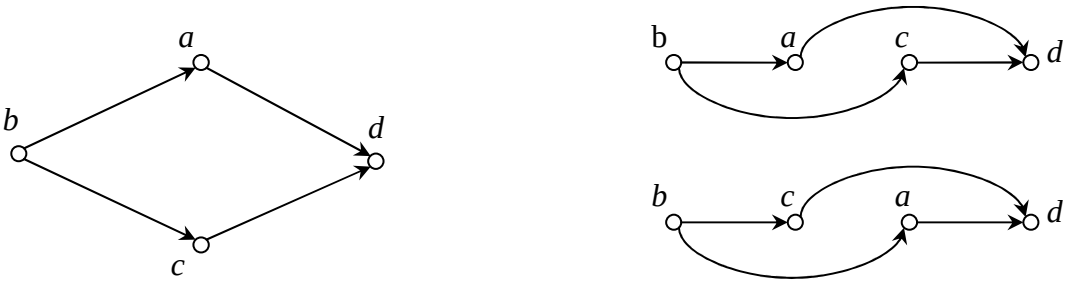
Топологічне сортування графу можна розглядати як таке впорядкування його вершин уздовж горизонтальної лінії, що всі ребра спрямовані зліва направо. Топологічне сортування має багато практичних застосувань, серед яких, наприклад, впорядкування задач в певному робочому плані.

Відзначимо важливість того факту, що граф повинен бути ациклічним. В іншому випадку, коли в графі є хоча б один орієнтований цикл (або контур), то в такому графі топологічне сортування неможливе. Це стверджує наступна теорема.

Теорема 29.2. Якщо в орієнтованому графі існує хоча б один цикл, то в ньому не існує топологічного сортування.

Доведення. Припустимо, що в графі існує пара вершин v та u, такі що існує орієнтований маршрут з v до u та, навпаки, – з u до v. Зрозуміло, що в такому випадку ці вершини будуть включені в єдиний орієнтований цикл. При спробі побудувати топологічне сортування, з того факту, що є маршрут з v до u, вершина v повинна передувати вершині u. Водночас, з факту існування маршруту з u до v слідує, що вершина u повинна передувати вершині v. Але це неможливо одночасно. Отже, топологічне сортування в графі з орієнтованим циклом не існує. ►

В загальному випадку, топологічне сортування, якщо воно існує, то воно не обов'язково буде єдиним. Наприклад, для графу на рис. 29.7, *а* існує два можливих впорядкування вершин в рамках топологічного сортування (рис. 29.7, *б*).



а) б)

Рис. 29.7. Приклад топологічного сортування.

Рекурсивний варіант процедури пошуку вглиб дозволяє досить просто побудувати топологічне сортування у ациклічному графі. В наведеному нижче псевдокоді представлені дві процедури: TopologicalSort(*G*) та DFSR(*G*, *s*). Причому DFSR є аналогічною до наведеної у лістингу 29.4 окрім невеликих нововведень, які, власне, й призвані реалізувати топологічне сортування. Для кожної вершини *v* вводиться параметр *f*[*v*] – номер вершини у топологічному сортуванні. У лістингу 29.5 змінна *current_label* містить поточний номер у топологічному сортуванні, який буде приписано черговій розглянутій вершині.

```
TopologicalSort(граф G)
1. позначити всі вершини як не відвідані
2. current_label ← n (кількість вершин графу)
3. для кожної вершини v графу G:
4.         if вершина v ще не відвідана
5.                 DFSR(G, v)

DFSR(граф G, початкова вершина s)
1. позначити s як відвідану
2. для кожного ребра (s, u) в G:
3.         if вершина u ще не відвідана
4.                 DFSR(G, u)
5. f[s] ← current_label
6. current_label ← current_label – 1
```

Лістинг 29.5. Алгоритм топологічного сортування на основі пошуку вглиб.

Продемонструємо виконання процедури TopologicalSort для графу, представленого на рис. 29.7, *a*. Нехай, вершини в циклі, який міститься в рядку 3 процедури TopologicalSort, переглядаються в порядку: *a*, *b*, *c*, *d*. Тоді процедура DFSR у рядку 5 процедури TopologicalSort буде викликана двічі: для вершин *a* та *b*. Дійсно, під час першого виклику DFSR(*G*, *a*) призведе до того, що вершини *a* та *d* отримають номери *f*[*a*] = 3 та *f*[*d*] = 4, причому ці вершини вже будуть помічені як відвідані. Наступний виклик – DFSR(*G*, *b*) призначить номери вершинам *b* та *c*: *f*[*b*] = 1 та *f*[*c*] = 2. Зрозуміло, що різні варіанти топологічного сортування обумовлені різним порядком перебору вершин у циклі в рядку 3 процедури TopologicalSort та ребер у рядку 2 процедури DFSR.

Час роботи процедури TopologicalSort дорівнює $O(n + m)$, де n і m – відповідно кількість вершин та ребер графу *G*. В цьому легко пересвідчитись, адже дана процедура потребує O(1) часу для обробки однієї вершини та O(1) часу для обробки одного ребра.

29.7. Підрахунок сильних компонент зв’язності

У розділі 29.4 розглядався метод визначення компонент зв’язності у неорієнтованому графі. Як відомо з теми 26, в орієнтованих графах вводиться декілька понять зв’язності,

залежно від степеню: сильна зв'язність, однобічна зв'язність і слабка зв'язність. Поняття компонент зв'язності у неорієнтованому графі можна перенести на компоненти слабкої зв'язності орієнтованого графу. Тут ми розглянемо сильно зв'язані компоненти – підграфи сильно зв'язаного орієнтованого графу, які володіють наступною властивістю.

Означення 29.3. **Сильною компонентою зв'язності** орієнтованого графу $G = (V, E)$ є максимальна множина вершин $C \subseteq V$, така що для кожної пари вершин u та v з C існує орієнтований маршрут (шлях) як з u до v, так і з v до u.

На рис. 29.8, *а* наведені приклади сильно зв'язаних компонент (позначені пунктиром).

Легко показати, що аналогічно до компонент зв'язності неорієнтованого графу, сильні компоненти зв'язності утворюють класи еквівалентності відношення сильної зв'язності.

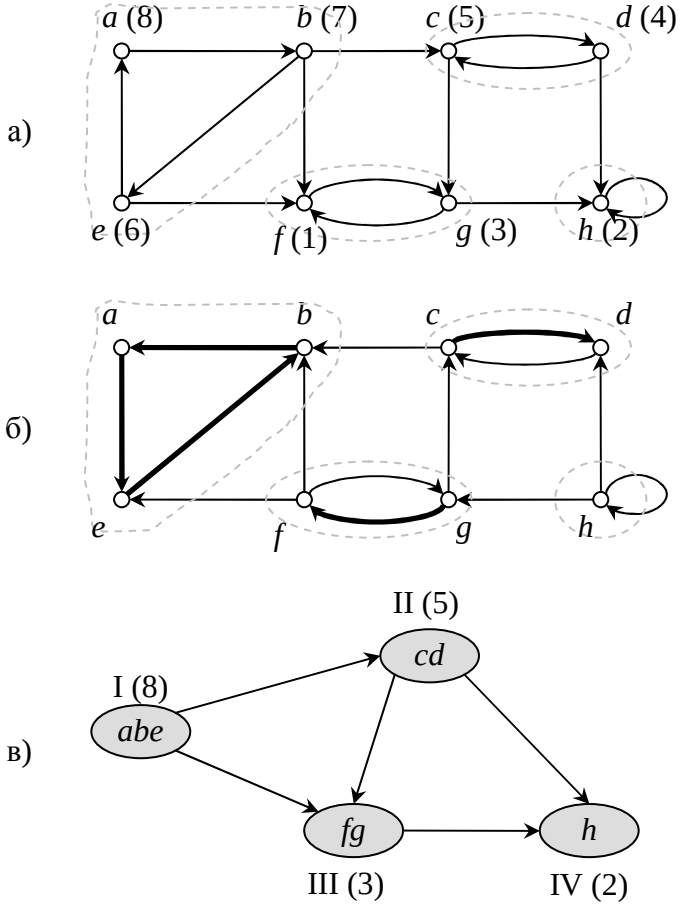


Рис. 29.8. Сильно зв'язані компоненти.

Алгоритм пошуку сильно зв'язаних компонент графу $G = (V, E)$ використовує транспонування графу G, яке визначається як граф $G^T = (V, E^T)$, де $E^T = \{(u, v) \mid (v, u) \in E\}$, тобто E^T складається з ребер G, але зі зміненою орієнтацією кожного ребра на зворотну. Слід зазначити, що графи G та G^T мають одні й ті самі сильно зв'язані компоненти: u та v досяжні одна з одної в G тоді й тільки тоді, коли вони досяжні одна з одної в G^T. На рис. 29.8, *б* показаний граф, який представляє собою транспонування графу з рис. 29.8, *а*.

Нижче наводиться алгоритм, який за лінійний час $O(n + m)$ знаходить сильні компоненти зв'язності орієнтованого графу завдяки подвійному пошуку вглиб: одному – в графі G, та другому – в графі G^T. Процедура пошук вглиб, яка тут використовується, аналогічна до тієї, що використовувалась при топологічному сортуванні, окрім значень параметру $f[v]$. На цей раз даний параметр відіграє роль лічильнику часу і вказує на момент часу, коли робота алгоритму пошуку вглиб закінчилась у вершині v (тобто, коли були переглянуті всі інцидентні ребра вершини v). Значення параметру f для графу на рис. 29.8,*а* відображаються у дужках біля кожної вершини. Ці значення отримуються, якщо спочатку

виконати пошук вглиб з вершини *c* (в цьому випадку будуть досягнуті вершини *f*, *h*, *g*, *d*, *c*), а потім ще раз одну ітерацію пошуку вглиб від вершини *a*.

`StronglyConnectedComponents`(граф *G*)
1. Викликати DFS_Loop(*G*) для обчислення часу звершення *f*[*v*] для кожної вершини *v*
2. Побудова транспонованого графу G^T
3. Викликати DFS_Loop(G^T), проте в головному циклі вершини розглядаються в порядку зменшення значень *f*[*v*], які були обчислені в рядку 1
4. Дерева покриття, які були отримані при пошуку вглиб у рядку 3, є сильними компонентами зв'язності

`DFS_Loop`(граф *G*)
1. позначити всі вершини як не відвідані
2. $t \leftarrow 0$
3. для кожної вершини *v* графу *G*:
4. `if` вершина *v* ще не відвідана
5. DFSR(*G*, *v*)

`DFSR`(граф *G*, початкова вершина *s*)
1. позначити *s* як відвідану
2. для кожного ребра (*s*, *u*) в *G*:
3. `if` вершина *u* ще не відвідана
4. DFSR(*G*, *u*)
5. $t \leftarrow t + 1$
6. $f[s] \leftarrow t$

Лістинг 29.6. Алгоритм визначення сильних компонент зв'язності.

Ідея, яка лежить у наведеному алгоритмі, спирається на ключову властивість графу компонентів $G^{SCC} = (V^{SCC}, E^{SCC})$, який визначається наступним чином.

Означення 29.4. Нехай граф $G = (V, E)$ має сильно зв'язані компоненти $C_1, C_2, \ldots, C_k$. Множина вершин **графу компонент** $V^{SCC} = \{v_1, v_2, \ldots, v_k\}$ складається з вершин v_i для кожної сильно зв'язаної компоненти C_i графу *G*. Якщо в графі є ребро (*u*, *w*) для деяких двох вершин $u \in C_i$ та $w \in C_j$, то в графі компонент є ребро $(v_i, v_j) \in E^{SCC}$.

Граф компонент для графу з рис. 29.8, *а* показаний на рис. 29.8, *в*.

Ключовою властивістю графу компонентів є те, що він представляє собою орієнований ациклічний граф, як слідує з наведеної нижче леми.

Лема 29.5. Нехай *C* та *C′* – різні сильно зв'язані компоненти орієнтованого графу $G = (V, E)$, і нехай $u, v \in C$ та $u', v' \in C'$, а крім того, припустимо, що в *G* є шлях з *u* до *u′*. В такому випадку в *G* не може бути шляху з *v′* до *v*.

При розгляду вершин в процесі другого пошуку вглиб у порядку зменшення часу завершення роботи вершин, який був обчислений при першому запуску пошуку вглиб, ми по суті відвідуємо вершини графу компонентів (кожна з яких відповідає сильно зв'язаній компоненті *G*) у порядку топологічного сортування.

Наступна лема та наслідок з неї описують ключову властивість, яка пов'язує сильно зв'язані компоненти та час завершення, який отриманий під час першого пошуку вглиб.

Лема 29.6. Нехай *C* та *C′* – різні сильно зв'язані компоненти орієнтованого графу $G = (V, E)$. Припустимо, що є ребро $(u, v) \in E$, де $u \in C$ та $v \in C'$. Тоді $f(C) > f(C')$, де $f(C)$ – це максимальний час завершення будь-якої вершини з компоненти *C*.

Наслідок. Нехай *C* та *C′* – різні сильно зв'язані компоненти орієнтованого графу $G = (V, E)$. Припустимо, що є ребро $(u, v) \in E^T$, де $u \in C$ та $v \in C'$. Тоді $f(C) < f(C')$.

Цей наслідок дає нам ключ до розуміння того, як працює процедура StronglyConnectedComponents. Подивимось, що відбувається, коли виконуємо другий пошук

углиб над графом G^T. Ми починаємо з сильно зв'язаної компоненти C, яка має найбільший час завершення $f(C)$. Пошук починається з деякої вершини $x \in C$, при цьому відвідуються всі вершини компоненти C. Відповідно зазначеному вище наслідку, у G^T немає ребер від C до іншої сильно зв'язаної компоненти, тож при пошуку з x не відвідується жодна вершина з інших компонент. Відповідно, дерево покриття, коренем якого є x, містить тільки вершини з C (ребра дерев покриття на рис. 29.8, *б* для графу G^T позначені потовщеними лініями). Після того, як будуть відвідані всі вершини в C, пошук в рядку 3 процедури StronglyConnectedComponents обирає в якості кореня вершину з деякої іншої сильно зв'язаної компоненти C', час завершення якої найбільший серед усіх інших компонент, окрім C. Тепер пошук відвідує всі вершини в C'.

В загальному випадку, коли пошук углиб в G^T відвідує деяку сильно зв'язану компоненту, всі ребра, які виходять з цієї компоненти, йдуть у вже опрацьовані компоненти. Відповідно, кожний пошук углиб опрацьовує тільки рівно одну компоненту. На основі даних міркувань отримуємо наступну теорему.

Теорема 29.3. Процедура StronglyConnectedComponents коректно обраховує сильно зв'язані компоненти орієнтованого графу G.

На рис. 29.8, *в* значення $f(C)$ для компонент показані в дужках біля римських чисел, які, в свою чергу, вказують на порядковий номер обходу сильно зв'язаної компоненти під час другого пошуку вглиб.