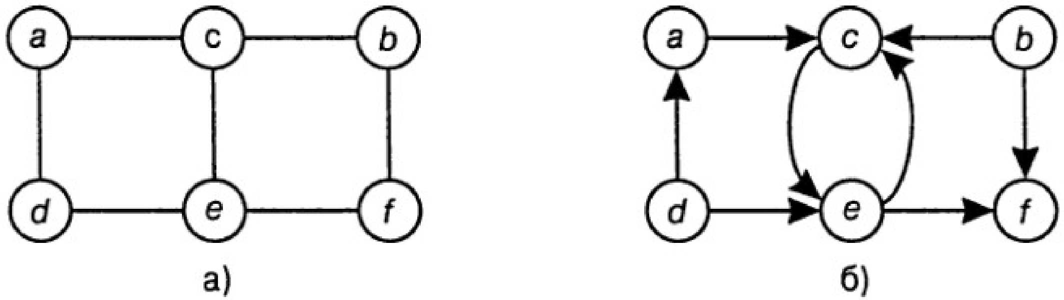
Алгоритми та складність

II семестр

Лекція 8

Графи

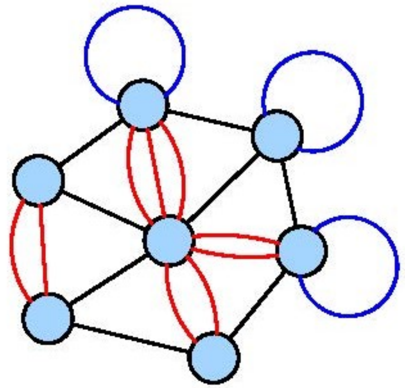
- *Граф* (нестрого) – сукупність вершин на площині, що з'єднані ребрами (чи дугами).
- *Граф* (строго): пара множин G = (V, E), де V – множина вершин, E – множина ребер.
- У псевдокоді для графа G позначаємо ці множини G.V та G.E.
- *Неорієнтований* граф: пари вершин неупорядковані.
- *Орієнтований* граф (орграф): пари вершин впорядковані.



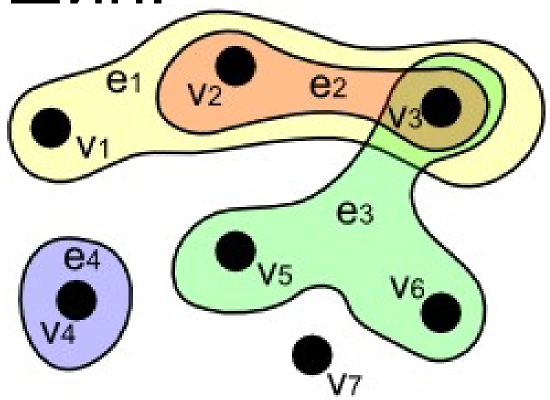
- *Петлі* – це ребра, які беруть початок і закінчуються в одній вершині; якщо явно не вказано інакше, вважаємо, що граф не містить петель. Неорієнтовані графи не можуть мати петель.

Графи

- *Мультиграф* допускає наявність декількох ребер, що з'єднують пару вершин (кратні ребра), а також петель (псевдограф).



- *Гіперграф* містить *гіперребра*, які можуть з'єднувати довільну кількість вершин.



- Ряд алгоритмів на графах можливо узагальнити на ці графовидні структури.

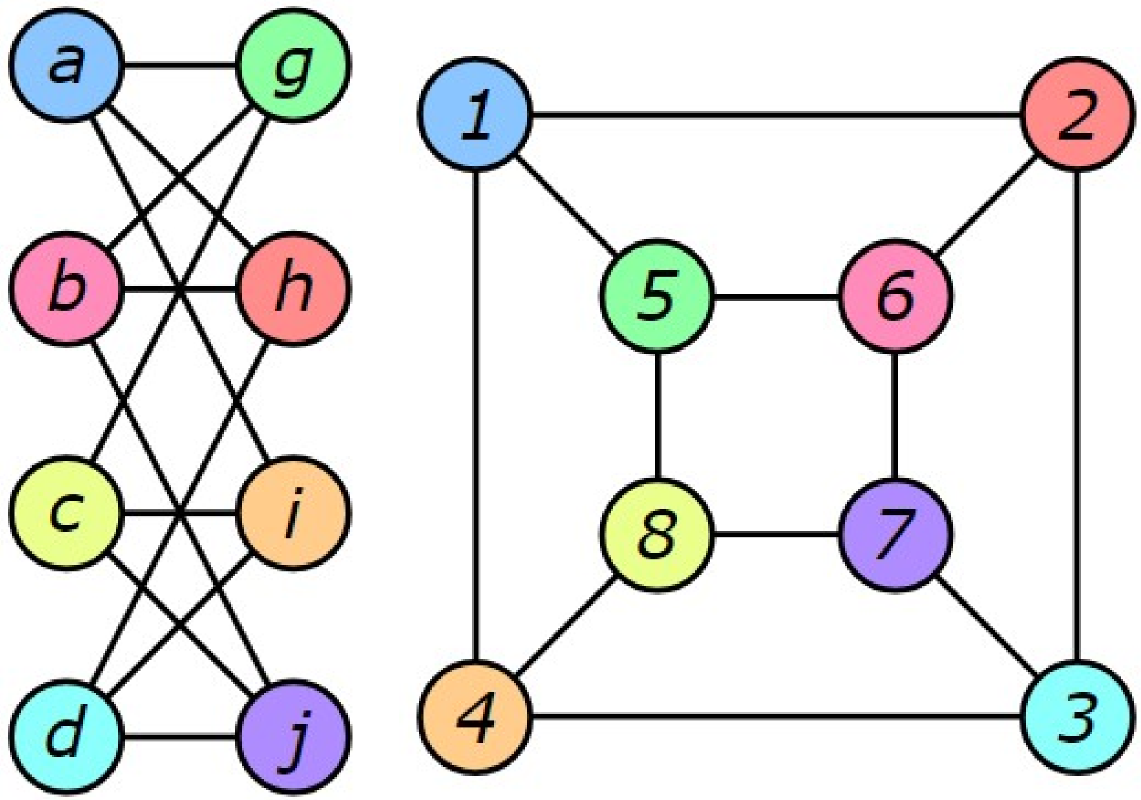
Графи

- Два графи G = (V, E) та G1 = (V1, E1) *ізоморфні*, якщо існує бієкція f : V→V1 така, що

$$(u,v)\in E \Leftrightarrow (f(u), f(v))\in E1.$$

- Тобто, можна перенумерувати (переіменувати) вершини, не чіпаючи ребра.

f(a) = 1
f(b) = 6
f(c) = 8
f(d) = 3
f(g) = 5
f(h) = 2
f(i) = 4
f(j) = 7



Графи

- Неорієнтований граф за визначенням не може містити між парою вершин більше одного ребра.
- Можна оцінити кількість ребер для неорієнтованого графа без петель:

$$0 \leq |E| \leq |V|(|V| - 1)/2.$$

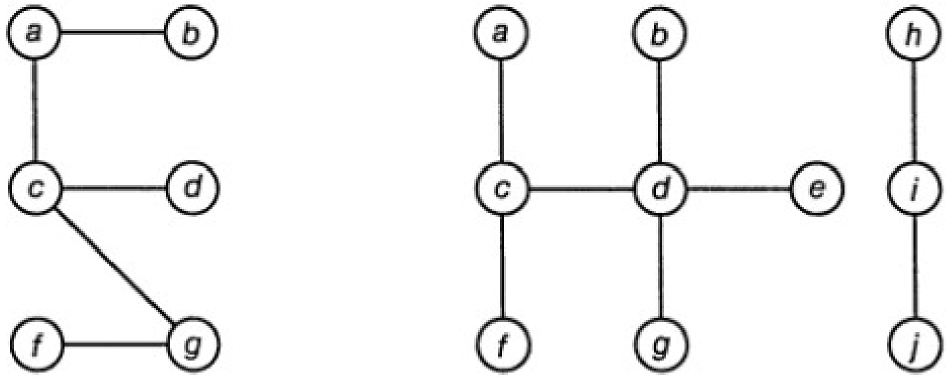
- Ділимо на 2, бо через неорієнтованість кожне ребро враховується двічі.
- Граф *повний*, якщо в нього кожна пара вершин з’єднана ребром.
- Граф *щільний* (dense), якщо кількість ребер у ньому близька до $|V|^2$.
- Граф *розріджений* (sparse), якщо він має відносно небагато ребер (відчутно менше за $|V|^2$).
- Далі в асимптотичних позначеннях для зручності кількість ребер і вершин позначатимемо просто E і V.

Графи

- *Шлях* (*маршрут*) від вершини u до вершини v – послідовність суміжних вершин, що починається з вершини u і закінчується у вершині v.
- Шлях *простий*, якщо всі ребра у ньому різні.
- Довжина шляху – кількість ребер в ньому.
- Граф *зв'язний*, якщо для довільної пари його вершин u та v існує шлях з u в v.
- Незв'язний граф складається зі *зв'язних компонент* – максимальних зв'язних підграфів графа, які не можна розширити включенням додаткової вершини, суміжної з якоюсь з його вершин.
- *Цикл* – простий шлях додатної довжини, що починається і завершується в тій самій вершині.
- Граф *ациклічний*, якщо він не містить циклів.

Графи

- *Вільним деревом* називають зв'язний ациклічний граф.
- Граф, що не містить циклів, але не обов'язково зв'язний – *ліс*. Кожна компонента зв'язності лісу є деревом.



- Для дерев справедливо $|E| = |V| - 1$.
- Для зв'язних графів цієї умови достатньо для визначення наявності циклу.
- Вільне дерево можна перетворити в *кореневе*, призначивши корінь і зорієнтувавши інші вершини.

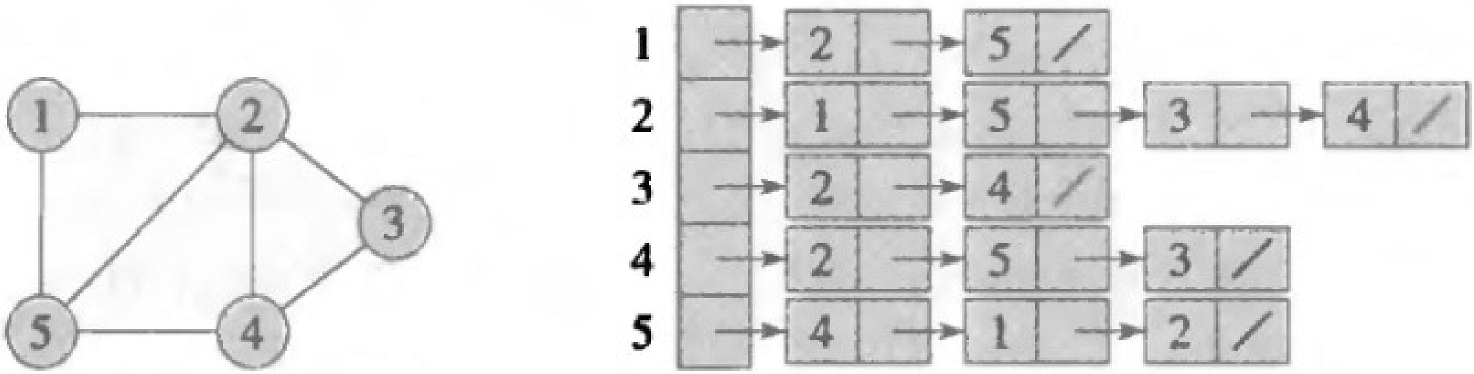
Представлення графів

- Існує два основних способи представлення графів: *списки суміжності* та *матриця суміжності*.
- Обидва підходять як для орієнтованих, так і для неорієнтованих графів.

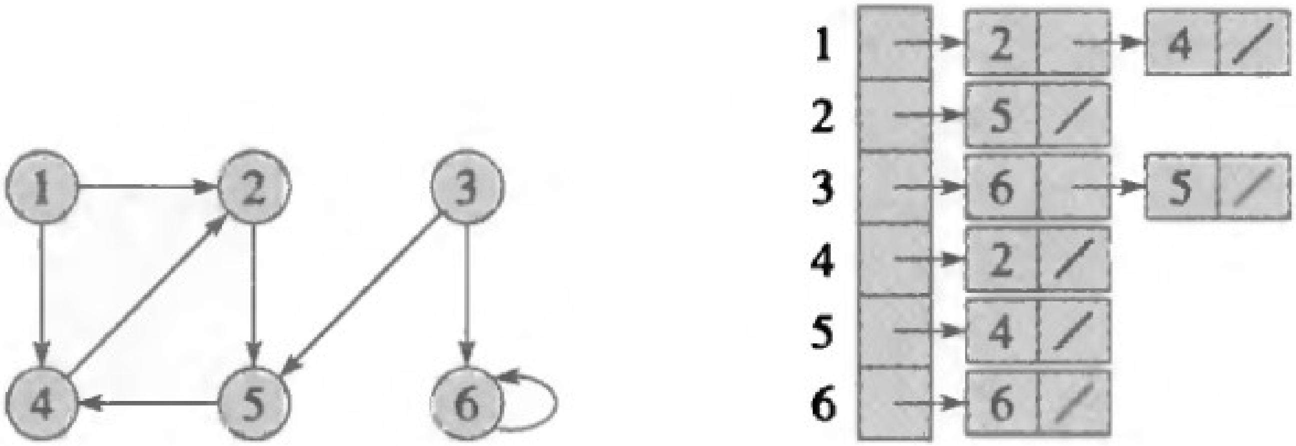
<u>Списки суміжності</u>

- Масив *Adj* з |V| списків для кожної вершини V.
- Для кожної вершини $u \in V$ список суміжності *Adj*[*u*] містить всі вершини, суміжні з u в графі G.
- Для орієнтованого графа сума довжин усіх списків суміжності дорівнює |E|, для неорієнтованого – 2|E|, бо кожне ребро (u,v) фігурує у списках для u та v.
- Використана пам'ять Θ(V+E).
- Для зважених графів вага w(u,v) ребра (u,v) зберігається разом з вершиною v в списку для u.
- Недолік: щоб перевірити наявність ребра у графі, потрібно проводити пошук по списку.

Представлення графів



Представлення неорієнтованого графа через списки суміжності



Представлення орієнтованого графа через списки суміжності

Представлення графів

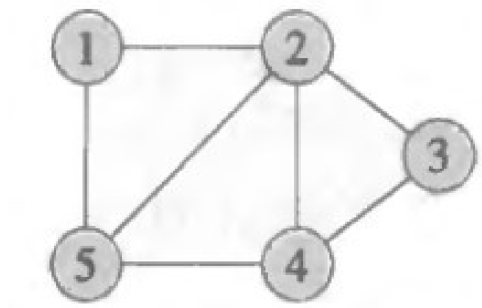
Матриця суміжності

- Нехай вершини пронумеровані числами від 1 до |V|, тоді кожен елемент a_{ij} матриці суміжності такий:

$$a_{ij} = \begin{cases} 1, \text{якщо } (i,j) \in E, \\ 0, \qquad\qquad \text{інакше.} \end{cases}$$

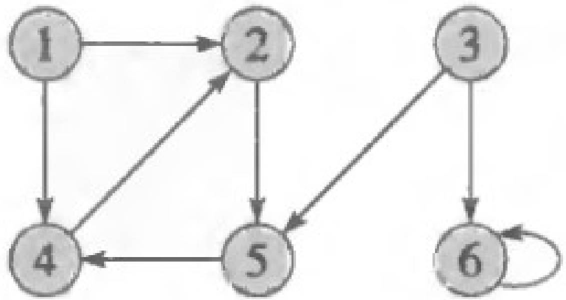
- Пам’ять незалежно від кількості ребер $\Theta(V^2)$.
- Для неорієнтованих графів матриця суміжності A симетрична відносно головної діагоналі ($A = A^T$), що часом дозволяє зберігати лише ті елементи, які розташовані на головній діагоналі і вище.
- Для зважених графів вага w(u,v) ребра (u,v) зберігається в елементі a_{ij}. Якщо відповідне ребро відсутнє – значення NIL (для багатьох алгоритмів зручніше використовувати 0 чи ∞).
- Для незважених графів для представлення одного ребра буде достатньо 1 біта, що зекономить пам’ять.

Представлення графів



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Представлення неорієнтованого графа через матрицю суміжності (вона симетрична)



	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Представлення орієнтованого графа через матрицю суміжності

Представлення графів

Яке з представлень краще вибрати?

Списки суміжності:

• Найкраще підходять для розріджених графів (отже, і для загального випадку).

• Більшість алгоритмів намагаються використовувати саме таке представлення для вхідного графа.

Матриця суміжності:

• Найкраще підходить для щільних і повних графів.

• Якщо треба вміти швидко визначати, чи є ребро між двома вершинами.

• При роботі з невеликими графами.

Представлення графів

- В більшості алгоритмів вершини та/або ребра графа мають певні атрибути.
- В описі алгоритму достатньо позначень типу *v*.*d* (атрибут *d* вершини *v*) чи (*u*,*v*).*f* (атрибут *f* ребра (*u*,*v*)).
- Реалізація атрибутів вершин і ребер залежить від алгоритму, мови програмування, а також того, як інші частини програми використовують граф.
- Зокрема, при представленні через списки суміжності атрибути можуть зберігатися у додаткових масивах, паралельних *Adj* (наприклад, атрибут *u*.*d* міститься в елементі масива *d*[*u*]).
- Об’єктно-орієнтовані мови дозволяють природним чином реалізовувати атрибути вершин.

Пошук в ширину

- *BFS* (breadth-first search) – один з найпростіших алгоритмів обходу графа, його ідея використовується в ряді інших алгоритмів.
- Для заданого графа G=(V,E) та початкової вершини (джерела) s алгоритм обходить всі ребра G, «відкриваючи» всі досяжні з s вершини.
- Одночасно при цьому обчислюється відстань (мінімальна кількість ребер) від s до кожної досяжної з неї вершини.
- В ході обходу будується «дерево пошуку в ширину» з коренем s, яке містить всі досяжні з s вершини.
- Для кожної досяжної з s вершини v простий шлях у дереві відповідає «найкоротшому шляху» (містить найменшу кількість ребер) від s до v в графі G.
- Алгоритм працює як для орієнтованих, так і для неорієнтованих графів.

Пошук в ширину

- Чому пошук «в ширину»: рухаємось як хвиля, щоб розпочати пошук вершин на відстані (k+1), маємо обійти всі вершини на відстані k.
- Алгоритм фарбує вершини в три кольори: білий, сірий, чорний.
- Спочатку всі вершини *білі*.
- Коли вершина вперше знайдена (*відкривається*), вона розфарбовується – стає *сірою*, а потім *чорною*.
- Якщо $(u,v) \in E$ та вершина u *чорна*, то v має бути *сірою* або *чорною*, тобто всі вершини, суміжні з чорною, вже відкриті.
- *Сірі* вершини є ніби границею між відкритими і невідкритими вершинами, вони можуть мати *білих* сусідів.

Пошук в ширину

- Алгоритм працює над вхідним графом у формі списків суміжності.
- Кожна вершина має додаткові атрибути кольору *color* та вершини-попередника π (якщо u не має попередника (u=s чи u не відкрита), то u.π=NIL).
- Відстань від s до вершини u зберігається в u.d.
- Для роботи з сірими вершинами – черга Q.
- Процедура пошуку в ширину будує дерево, що спочатку містить одну вихідну вершину-корінь s.
- Якщо в процесі сканування сусідів вершини u відкривається біла вершина v, то вершина v і ребро (u,v) додаються до дерева.
- При цьому говоримо, що u є *попередником* (*батьком*) v в дереві пошуку в ширину.
- Кожна вершина може бути відкритою не більше ніж раз, тому вона матиме не більше одного батька.

Пошук в ширину

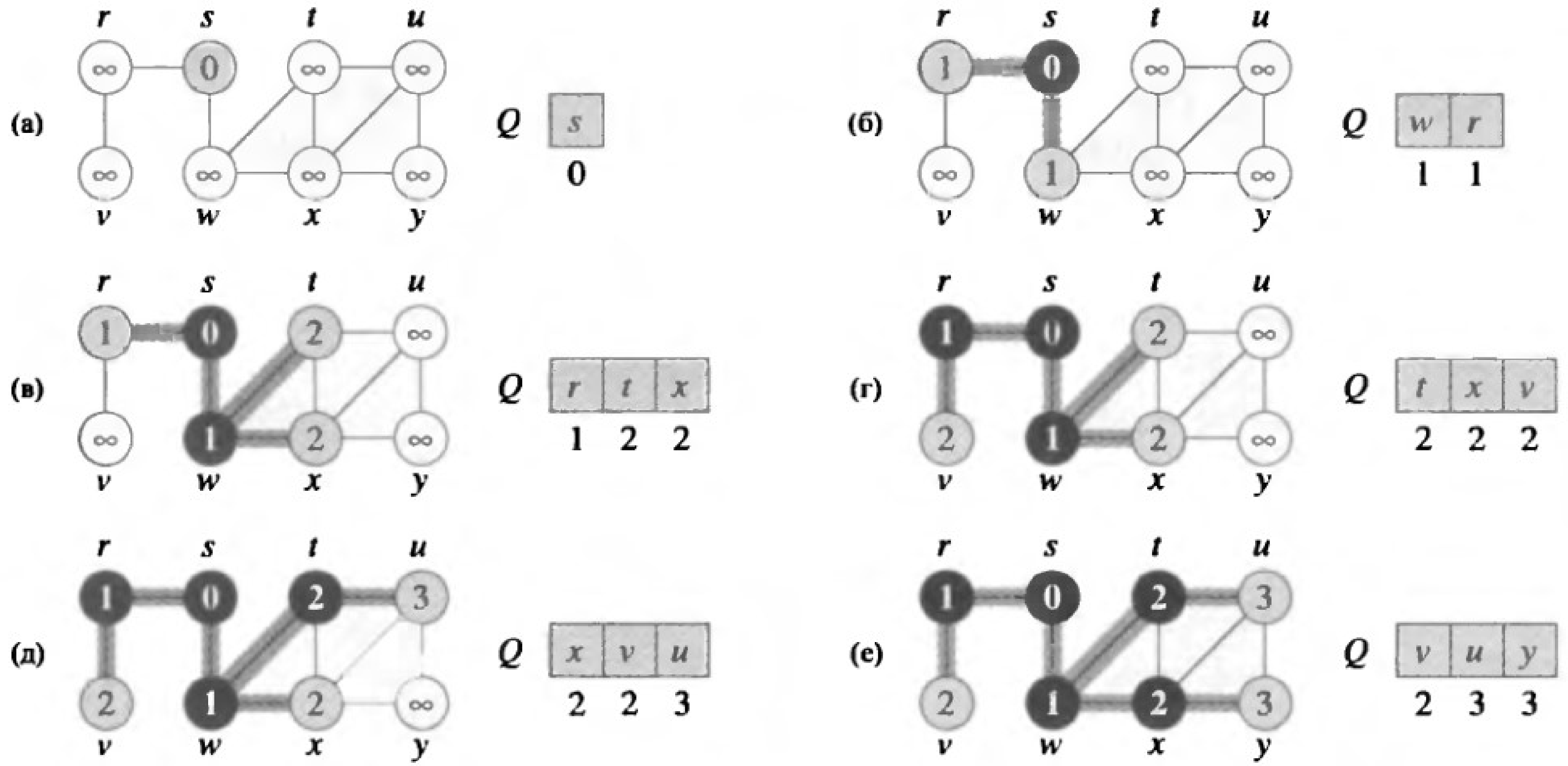
```
BFS(G, s)
for Каждой вершины u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for Каждой вершины v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

Всі вершини крім початкової фарбуються в білий колір, виставляються початкові значення ∞ для відстані і NIL для батька.

Обробляється початкова вершина s та вставляється в чергу.

Для кожної вершини u з черги скануються її сусіди.
Якщо сусід v був невідвіданий, йому призначається сірий колір, встановлюється відстань і вказується батько u.
Далі v поміщається в чергу.
Після перегляду всіх сусідів вершина u оброблена і стає чорною.

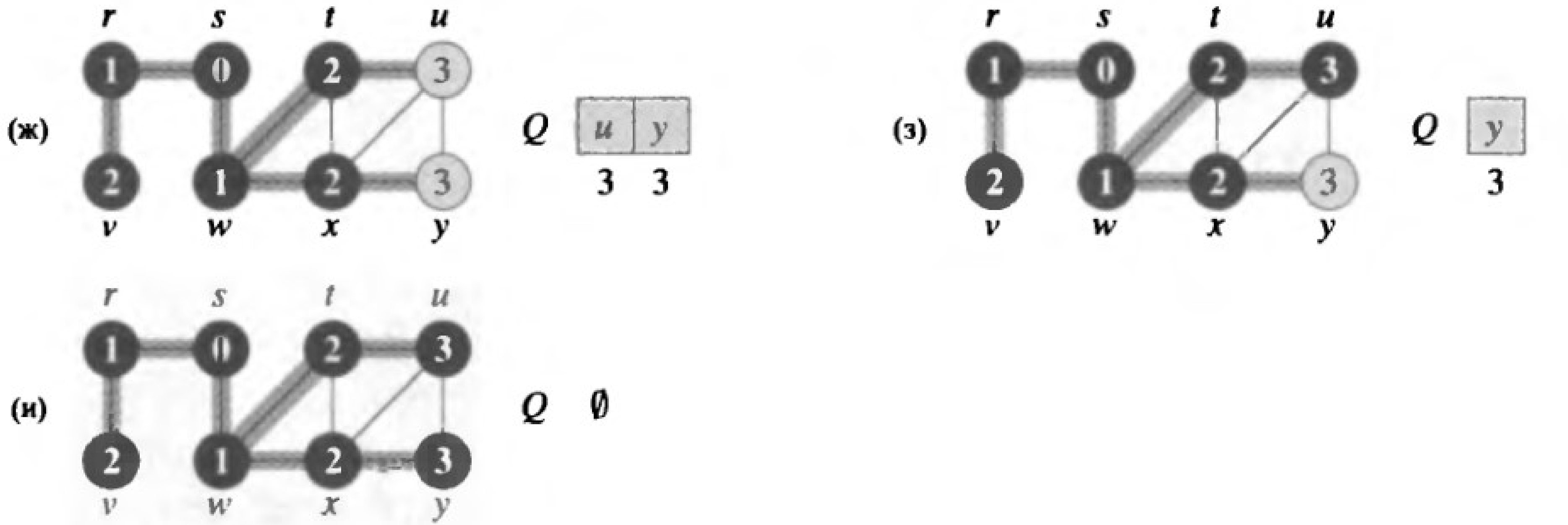
Пошук в ширину



Приклад роботи BFS для неорієнтованого графа.

Ребра, що додаються до дерева, заштриховані. В кожній вершині вказано атрибут відстані u.d. Показано стан Q на початку чергової ітерації роботи над чергою. В черзі під вершинами вказана відстань до них.

Пошук в ширину



Приклад роботи BFS для неорієнтованого графа – далі.

Ребра, що додаються до дерева, заштриховані. В кожній вершині вказано атрибут відстані u.d. Показано стан Q на початку чергової ітерації роботи над чергою. В черзі під вершинами вказана відстань до них.

- Результат пошуку в ширину залежить від порядку перегляду вершин-сусідів. Дерево пошуку в ширину при цьому може варіювати, але обчислені відстані d збережуться, бо не залежать від порядку обходу.

Пошук в ширину

- Оцінимо час роботи алгоритму з використанням групового аналізу.
- Після ініціалізації (за час O(V)) жодна вершина не розфарбовується білим, тому перевірка на «білизну» в середині циклу гарантує, що кожна вершина вноситься і згодом видаляється з черги не більше одного разу.
- Операції додавання і видалення з черги відбуваються за O(1), тому загальний час роботи з чергою складає O(V).
- Кожен список суміжності переглядається лише при видаленні відповідної вершини з черги, отже кожен список проскановується максимум один раз.
- Сума довжин всіх списків суміжності Θ(E), тому сумарний час їх сканування O(E).
- Загальний час роботи алгоритму O(V+E).

Пошук в ширину та найкоротші шляхи

- *Довжина найкоротшого шляху* (в незваженому графі) $\delta(s,v)$ від s до v – мінімальна кількість ребер на будь-якому шляху з s до v.

Лема. Нехай G=(V,E) – (не)орієнтований граф та $s \in V$ – його довільна вершина. Тоді для будь-якого ребра $(u,v) \in E$ виконується $\delta(s,v) \leq \delta(s,u) + 1$.

Доведемо це. Якщо вершина u досяжна з s, то досяжна і вершина v.

При цьому найкоротший шлях з s в v не може бути довшим за найкоротший шлях з s в u, за яким йде ребро (u,v) – нерівність справджується.

Якщо ж вершина u недосяжна з s, то $\delta(s,u)=\infty$ і нерівність також виконується.

Пошук в ширину та найкоротші шляхи

Теорема. *Коректність пошуку в ширину*

Нехай G=(V,E) – (не)орієнтований граф та процедура BFS виконується з початкової вершини $s \in V$.

Тоді в ході процедури BFS відкриваються всі вершини $v \in V$, досяжні з s, а по завершенні BFS для всіх $v \in V$ виконується $v.d = \delta(s,v)$.

При цьому для всіх досяжних з s вершин $v \neq s$ одним з найкоротших шляхів з s до v є шлях з s до $v.\pi$, за яким йде ребро $(v.\pi, v)$.

- Слід зауважити, що в алгоритмі достатньо розглядати дві категорії вершин: відвідані (темні, помічаються при відкритті) та невідвідані (білі). Поділ на сірі та чорні вершини дозволяє покращити розуміння роботи алгоритму.

Дерева пошуку в ширину

- Побудоване в результаті процедури BFS дерево пошуку в ширину відповідає атрибутам π в кожній вершині.
- Для графа G та вихідної вершини s визначимо *підграф передування* $G_\pi = (V_\pi, E_\pi)$, де

$$V_\pi=\{v\in V\text{: } v.\pi \neq \text{NIL}\}\cup\{s\},$$

$$E_\pi=\{(v.\pi,v)\in E\text{: } v\in V_\pi - \{s\}\}.$$

- Підграф передування G_π є *деревом пошуку в ширину*, якщо V_π складається з вершин, досяжних з s, та якщо для всіх $v\in V_\pi$ в G_π існує єдиний простий шлях з s у v такий, що одночасно є найкоротшим шляхом з s у v в графі G.
- Ребра в E_π називаються *ребрами дерева*.

Дерева пошуку в ширину

- За деревом пошуку в ширину можна відтворити всі вершини в найкоротшому шляху з s в v:

```
PRINT-PATH(G, s, v)
if v == s
    print s
elseif v.π == NIL
    print "Путь из" s "в" v "отсутствует"
else PRINT-PATH(G, s, v.π)
    print v
```

- Час роботи процедури лінійно залежить від кількості вершин, що виводяться, оскільки кожен рекурсивний виклик виконується для шляху на одиницю коротшому на поточний.

Пошук в глибину

- *DFS* (depth-first search) – пошук в глибину, рухаємося «углиб» графа, наскільки можливо.
- Відвідуються всі ребра, що виходять з відкритої останньою вершини v, покидаючи її, коли не залишиться невідвіданих ребер, і робиться відкат у вершину, з якої була відкрита v.
- Так відбувається, поки не відкриємо всі вершини, досяжні з початкової, інакше вибираємо нову невідкриту вершину і рухаємося з неї.
- *Підграф передування* $G_\pi = (V_\pi, E_\pi)$ для DFS, де

$$E_\pi=\{(v.\pi, v):\ v \in V_\pi \text{ та } v.\pi \neq NIL\}.$$

- Він утворює *ліс пошуку в глибину*, бо може складатися з декількох *дерев пошуку в глибину*. Ребра E_π – ребра дерева.

Пошук в глибину

- Те, що в результаті пошуку в глибину отримуємо ліс, а внаслідок пошуку в ширину – дерево, пов'язано з типовими використаннями цих обходів.
- Пошук в ширину найчастіше зв'язаний з пошуком довжин найкоротших шляхів з заданої вершини та графом передування, тому розглядається єдина вершина-джерело.
- Пошук в глибину може служити частиною інших алгоритмів над графом.
- В теорії не заборонено обмежуватися однією вихідною вершиною при пошуку в ширину чи проводити пошук в глибину від єдиного джерела.

Пошук в глибину

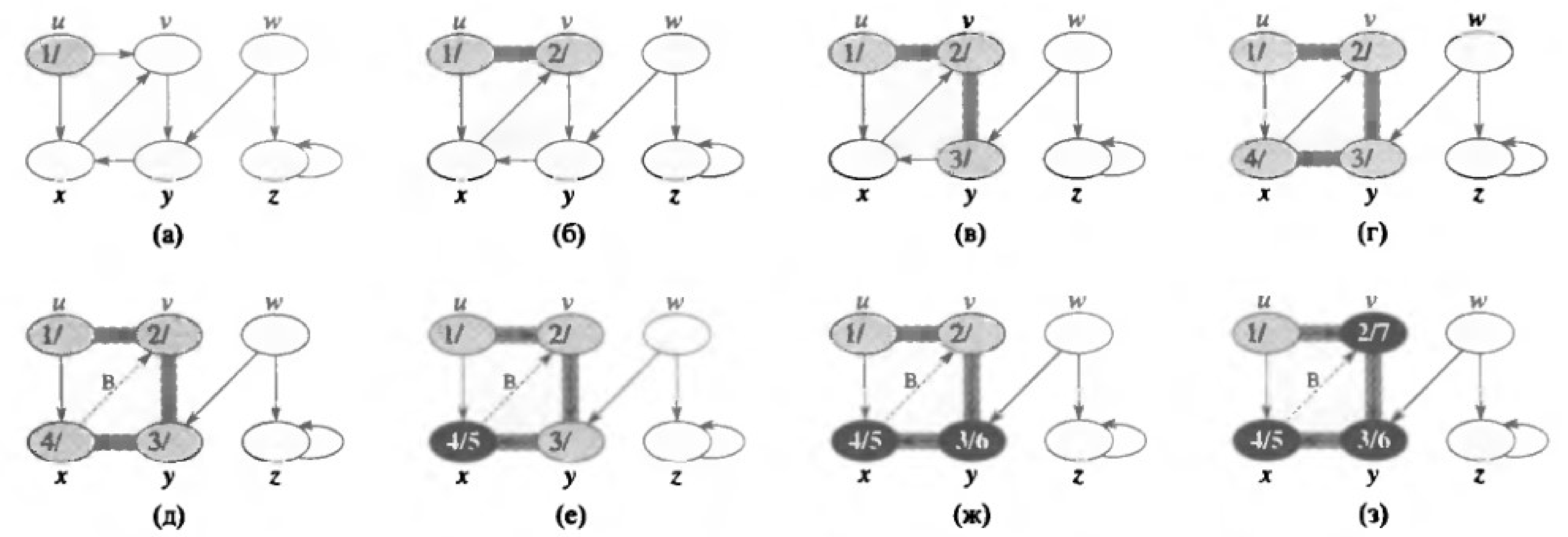
- Як і в пошуку в ширину, в процесі пошуку в глибину вершини розфарбовуються в три кольори: білий, сірий та чорний, що гарантує належність вершини лише до одного дерева пошуку в глибину.
- Також алгоритм проставляє у вершинах дві *мітки часу* (timestamp), першу v.d при відкритті вершини та другу v.f при завершенні роботи з вершиною.
- Мітки є цілими числами діапазону 1..2|V|, оскільки для кожної з |V| вершин є лише по одній події відкриття і закриття.
- Для кожної вершини u виконується u.d < u.f.
- До моменту u.d вершина u має білий колір, між u.d та u.f – сіра, а потім чорна.

Пошук в глибину

Алгоритм працює як з орієнтованими, так і з неорієнтованими графами

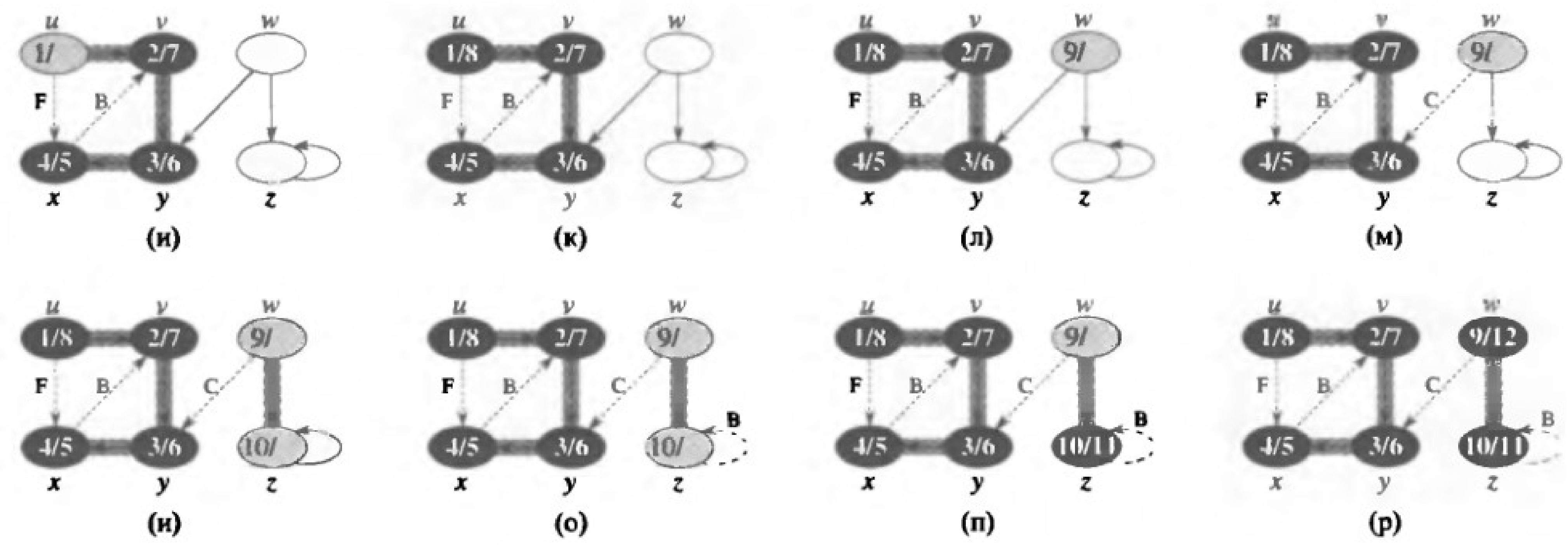
```
DFS(G)
for каждой вершины u ∈ G.V
        u.color = WHITE
        u.π = NIL
time = 0                                  // time – глобальна змінна
for каждой вершины u ∈ G.V
        if u.color == WHITE
                DFS-VISIT(G, u)

DFS-VISIT(G, u)
time = time + 1                           // відкрили білу вершину u
u.d = time
u.color = GRAY
for каждой v ∈ G.Adj[u]                   // дослідження ребра (u,v)
        if v.color == WHITE
                v.π = u
                DFS-VISIT(G, v)
u.color = BLACK                           // завершення роботи з u
time = time + 1
u.f = time
```

Пошук в глибину



DFS для орієнтованого графа. Досліджувані ребра затемнені, якщо це ребра дерева, і пунктирні інакше. Для ребер, які не є ребрами дерева є позначки B (back), C (cross), F (forward) для зворотних, перехресних та прямих ребер. У вершинах вказані мітки часу.

Пошук в глибину



DFS для орієнтованого графа. Досліджувані ребра затемнені, якщо це ребра дерева, і пунктирні інакше. Для ребер, які не є ребрами дерева є позначки B (back), C (cross), F (forward) для зворотних, перехресних та прямих ребер. У вершинах вказані мітки часу.

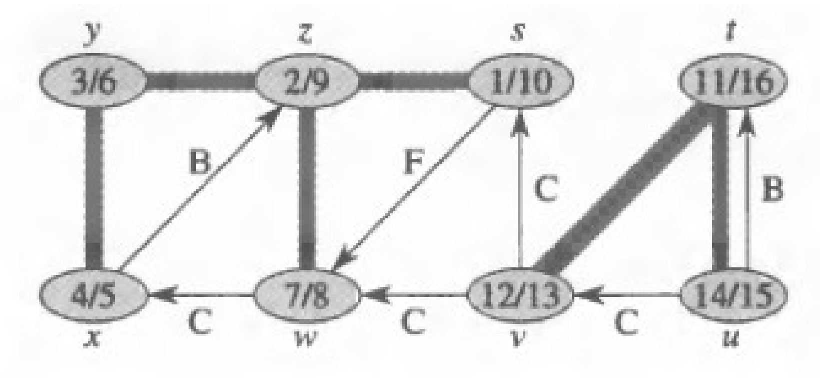
Пошук в глибину

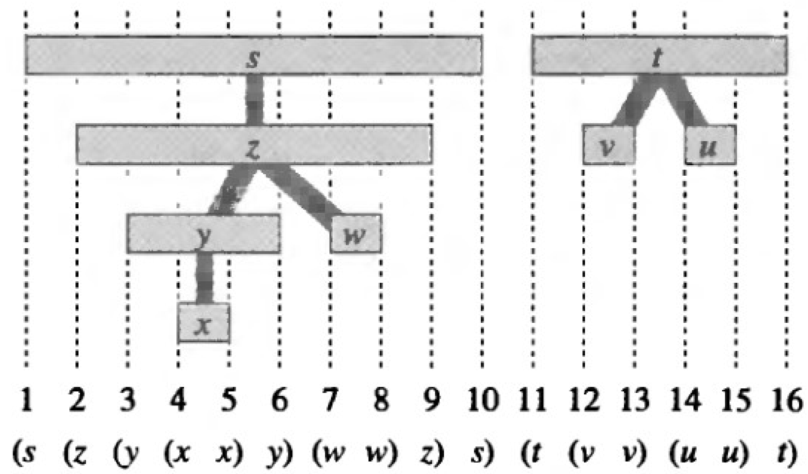
- Результат пошуку в глибину залежить від порядку розгляду вершин у рядку 5 процедури DFS та порядку відвідування суміжних вершин в рядку 4 DFS-VISIT.
- Будь-який результат пошуку може бути ефективно використаний при роботі інших алгоритмів і приведе до однакових по суті результатів.
- Час роботи DFS: Θ(V) плюс виклик внутрішньої процедури.
- Час роботи DFS-VISIT: викликається для кожної вершини рівно один раз, а цикл в рядках 4-7 – |Adj[v]| раз.
- Маємо $\sum_{v \epsilon V} |Adj[v]| = \Theta(E)$ і DFS-VISIT працює за Θ(E).
- Загальний час роботи алгоритму Θ(V+E).

Властивості пошуку в глибину

- Підграф передування G_{π} дійсно утворює ліс: структура дерев пошуку в глибину відображає структуру рекурсивних викликів DFS-VISIT. Тобто u = v.π $\Leftrightarrow$ виклик DFS-VISIT(G,v) відбувся при перегляді списку суміжності вершини u.
- Вершина v є потомком u в лісі пошуку в глибину $\Leftrightarrow$ вершина u була сірою в момент відкриття вершини v.
- Часи відкриття та закриття утворюють *дужкову структуру*. Якщо представити відкриття вершини u у вигляді дужки, що відкривається "(*u*", а її закриття як "*u*)", то послідовність відкриттів-завершень утворить коректний вираз в сенсі вкладеності дужок.

Властивості пошуку в глибину





- Інтервали між відкриттям та закриттям для кожної вершини відповідають вказаній дужковій структурі.
- Якщо два інтервали перетинаються, то один з них вкладений в інший, і вершина, яка відповідає меншому інтервалу, є потомком вершини, що відповідає більшому інтервалу.

Властивості пошуку в глибину

- <u>Теорема (про дужки)</u>. При довільному пошуку в глибину в (не)орієнтованому графі для будь-якої пари вершин u та v виконується одне з тверджень:
  - відрізки [u.d,u.f] та [v.d,v.f] не перетинаються, між вершинами u та v немає зв’язку типу предок-потомок;
  - відрізок [u.d,u.f] цілком міститься у відрізку [v.d,v.f], а вершина u є потомком v в дереві пошуку в глибину;
  - відрізок [v.d,v.f] цілком міститься у відрізку [u.d,u.f], а вершина v є потомком u в дереві пошуку в глибину.

Властивості пошуку в глибину

Доведемо її.

- Припустимо $u.d < v.d$.
- Розглянемо випадок $v.d < u.f$: вершину v відкрили, коли u залишалась сірою. Тобто, v є потомком u. Тому, перед поверненням до u, алгоритм дослідить всі вихідні з v ребра – отже відрізок $[v.d,v.f]$ повністю міститься у відрізку $[u.d,u.f]$.
- У випадку $u.f < v.d$ зі співвідношення часів відкриття та закриття вершин отримуємо $u.d < u.f < v.d < v.f$. Отже, відрізки $[u.d,u.f]$ та $[v.d,v.f]$ не перетинаються, а тому відкриття однієї вершини не відбудеться, поки інша сіра, так що жодна вершина не є потомком іншої.
- Випадок $v.d < u.d$ розглядається аналогічно (тут ролі u та v дзеркальні).

Властивості пошуку в глибину

- Наслідок (вкладеність інтервалів потомків). Вершина v є істинним (відмінним від самого u) потомком u в лісі пошуку в глибину в (не)орієнтованому графі G ⇔ $u.d < v.d < v.f < u.f$

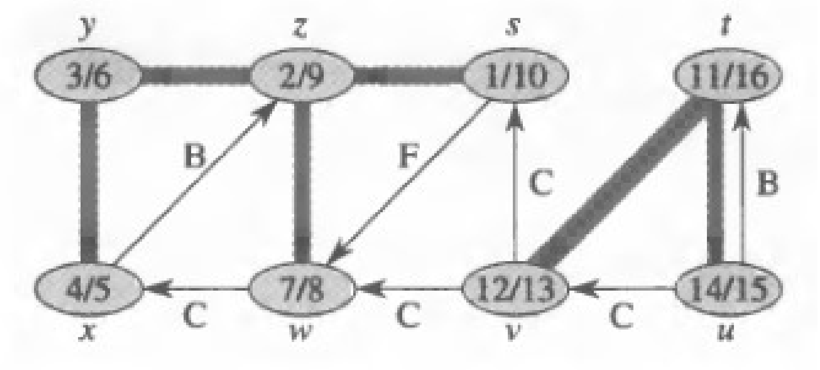
- Теорема (про білий шлях). В лісі пошуку в глибину в (не)орієнтованому графі вершина v є потомком u ⇔ в момент відкриття вершини u u.d вершина v досяжна з u по шляху, що складається лише з білих вершин.

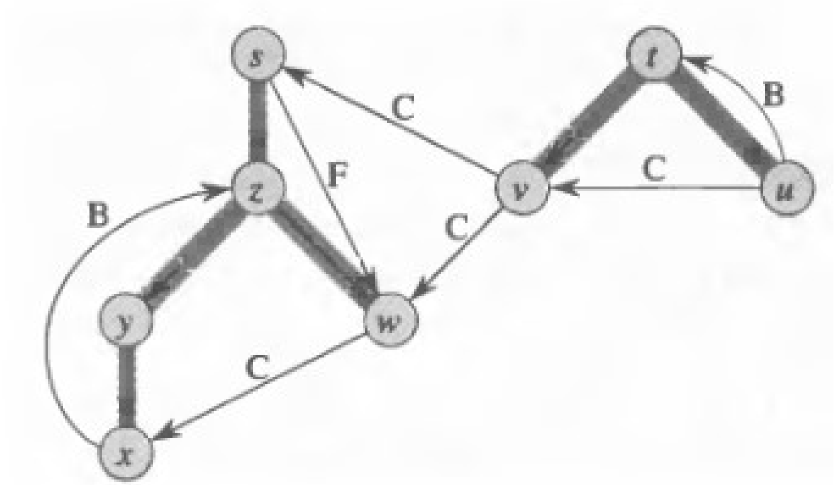
Класифікація ребер

- Класифікація ребер дозволяє отримати ряд інформації про граф.
- Типи ребер, отриманих при пошуку в глибину.

1. *Ребра дерева* – ребра лісу G_π. Ребро (u,v) є ребром дерева, якщо при його дослідженні була вперше відкрита вершина v.
2. *Зворотні ребра* – ребра (u,v), що з'єднують вершину u з її предком v в дереві пошуку в глибину. Петлі в орієнтованих графах також вважаються зворотними ребрами.
3. *Прямі ребра* – ребра (u,v), що не є ребрами дерева, і з'єднують u з її потомком v в дереві пошуку в глибину
4. *Перехресні ребра* – всі інші ребра графа. Немає зв'язку предок-потомок або з'єднуються вершини у різних деревах пошуку в глибину.

Класифікація ребер





- Будь-який граф можна зобразити так, щоб всі його прямі ребра і ребра дерев були направлені вниз, а зворотні ребра – вгору.

Класифікація ребер

- Алгоритм DFS можна модифікувати, щоб він класифікував ребра, які зустрічає.
- Кожне ребро (u,v) можна класифікувати за кольором вершини v при першому його відвідуванні (однак при цьому не розрізняються прямі і перехресні ребра):

1. WHITE. Ребро дерева (за визначенням).
2. GRAY. Зворотне ребро. Сірі вершини утворюють ланцюжок потомків; кількість сірих вершин на 1 більша за глибину останньої відкритої вершини в дереві; дослідження починається з найглибшої сірої вершини, тому ребро, що досягає іншої сірої вершини, досягає предка вихідної вершини.
3. BLACK. Пряме чи перехресне ребро. Ребро (u,v) пряме при $u.d < v.d$ та перехресне при $u.d > v.d$.

Класифікація ребер

- В неорієнтованому графі при класифікації ребер можлива певна неоднозначність, бо (u,v) та (v,u) по суті одне ребро.
- Тому класифікуємо ребро по *першій* категорії в списку можливих категорій для нього.
- Аналогічно, класифікацію можна виконувати відповідно до вигляду (u,v) чи (v,u), в якому ребро вперше зустрічається в процесі виконання алгоритму.

Теорема. При пошуку в глибину в неорієнтованому графі будь-яке його ребро буде або ребром дерева, або зворотним ребром.

- Таким чином, в неорієнтованому графі при пошуку в глибину ніколи не зустрінуться прямі чи перехресні ребра.

Класифікація ребер

Доведемо наведену теорему.

- Нехай (u,v) – довільне ребро графа та припустимо u.d < v.d.
- Тоді вершина v має бути відкрита та оброблена до того, як закриється (поки що сіра) вершина u – оскільки v знаходиться у списку суміжності u.
- Якщо ребро (u,v) досліджується в напрямку від u до v, то вершина v має бути на цей момент невідкритою, інакше ребро би вже дослідили в зворотній орієнтації. Отже, (u,v) стає ребром дерева.
- Якщо ж ребро (u,v) досліджується спочатку в напрямку від v до u, то воно буде зворотним, бо вершина u при першому дослідженні ребра сіра.

Класифікація ребер

- Дерево пошуку в ширину також можна використати для аналогічної класифікації досяжних з вихідної вершини ребер.

- При пошуку в ширину в *неорієнтованому* графі виконуються такі властивості (*довести*):
  - не існує прямих та зворотних ребер;
  - для кожного ребра дерева (u,v) справджується v.d = u.d + 1;
  - для кожного перехресного ребра (u,v) маємо v.d = u.d або v.d = u.d + 1.

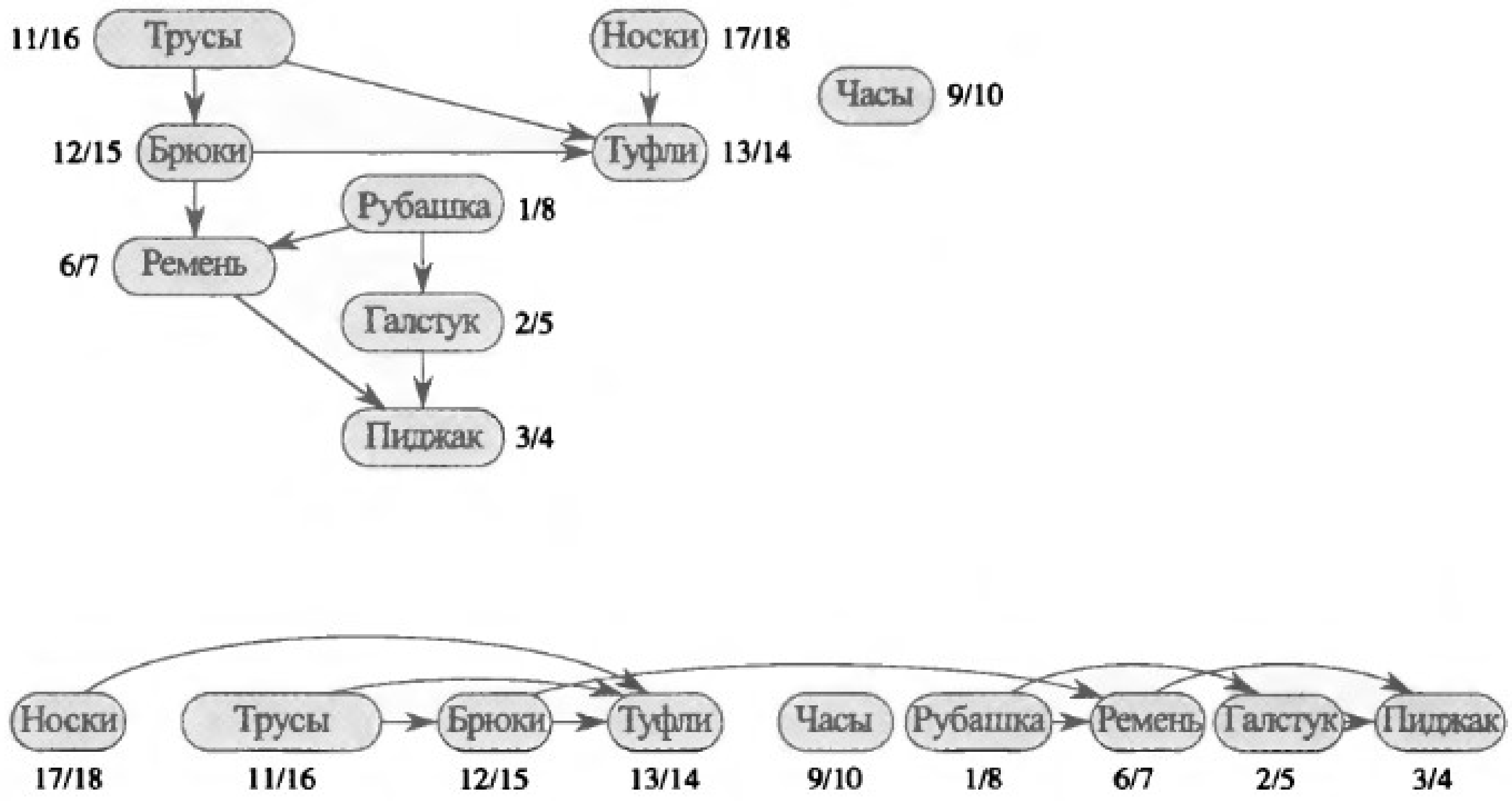
Класифікація ребер

- При пошуку в ширину в *орієнтованому* графі виконуються наступні властивості (*довести*):
  - не існує прямих;
  - для кожного ребра дерева (u,v) справджується v.d = u.d + 1;
  - для кожного перехресного ребра (u,v) маємо v.d ≤ u.d + 1;
  - для кожного зворотного ребра (u,v) отримуємо 0 ≤ v.d ≤ u.d.

Топологічне сортування

- *Топологічне сортування орієнтованого ациклічного графа* – лінійне впорядкування всіх його вершин таке, що якщо граф містить ребро (u,v), то вершина u зустрічається до вершини v.
- За наявності циклу таке впорядкування неможливе.
- Умовно, вершини графа упорядковуються вздовж горизонтальної лінії так, що всі ребра направлені зліва направо.
- Такі графи використовуються для вказування послідовності подій. Таким чином буде видно, які події обов'язково мають відбутися в певному порядку, а які можуть ставатися незалежно.

Топологічне сортування



Приклад топологічного сортування для ранкового одягання. Кожне орієнтоване ребро (u,v) означає, що річ u має бути одягнена перед річчю v.

Топологічне сортування

TOPOLOGICAL-SORT(G)

1 **Вызвать DFS(G) для вычисления времен завершения $v.f$ для каждой вершины v**

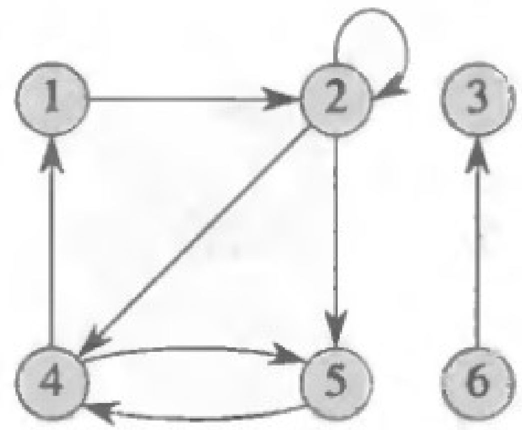
2 **По завершении работы над вершиной внести ее в начало связанного списка**

3 **return связанный список вершин**

- Можна помітити, що топологічно відсортовані вершини розташовані в порядку спадання часу завершення.
- Час роботи алгоритму Θ(V+E).
- Орієнтований граф ациклічний тоді і тільки тоді, коли пошук в глибину не знаходить в ньому зворотних ребер.

Сильно зв'язні компоненти

- Орієнтований граф *сильно зв'язний*, якщо будь-які його дві вершини досяжні одна з одної.
- Довільний орієнтований граф можна розбити на *сильно зв'язні компоненти* – класи еквівалентності відношення «бути взаємно досяжними».



Компоненти сильної зв'язності: {1,2,4,5}, {3} та {6}.

Сильно зв'язні компоненти

- Визначимо *граф компонентів* $G^{SCC} = (V^{SCC}, E^{SCC})$.
- Нехай граф G має сильно зв'язні компоненти $C_1, C_2, \ldots, C_k$.
- Множина вершин $V^{SCC} = \{v_1, v_2, \ldots, v_k\}$ містить вершину v_i для кожної сильно зв'язної компоненти C_i графа G.
- Якщо в G наявне ребро (x,y) для $x \in C_i$ та $y \in C_j$, то в графі компонентів є ребро $(v_i, v_j) \in E^{SCC}$.
- Тобто, якщо стиснути всі ребра між суміжними вершинами в кожній сильно зв'язній компоненті G, отримаємо граф G^{SCC}, вершинами якого і будуть згадані сильно зв'язні компоненти.
- Граф компонентів є ациклічним орієнтованим графом.

Сильно зв'язні компоненти

- Алгоритм пошуку сильно зв'язних компонент використовує транспонований граф

$$G^T = (V, E^T), \text{ де}$$

$$E^T = \{(u,v)\colon (v,u) \in E\}.$$

- Тобто E^T складається з ребер G з оберненою орієнтацією.
- За списками суміжності граф G^T отримується з графа G за час O(V+E).
- Графи G та G^T мають ті самі сильно зв'язні компоненти: вершини u та v досяжні одна з одної в G $\Leftrightarrow$ вони взаємно досяжні у G^T.

Сильно зв'язні компоненти

STRONGLY-CONNECTED-COMPONENTS(G)

1 Вызов DFS(G) для вычисления времен завершения $u.f$
для каждой вершины u
2 Вычисление G^{T}
3 Вызов DFS(G^{T}), но в основном цикле процедуры DFS
вершины рассматриваются в порядке убывания значений $u.f$,
вычисленных в строке 1
4 Вывод вершин каждого дерева в лесу поиска в глубину,
полученного в строке 3, в качестве
отдельного сильно связного компонента

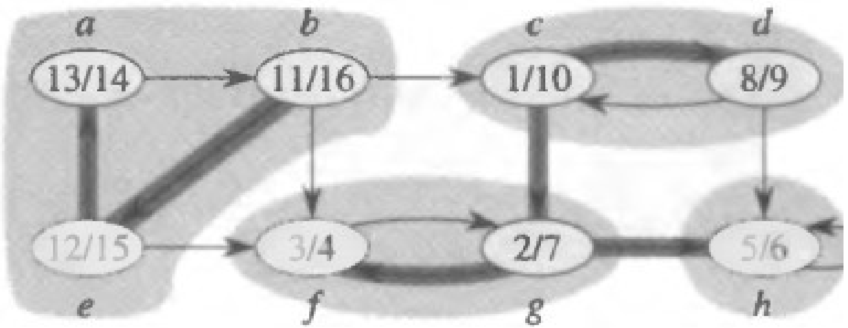
- Алгоритм знаходить сильно зв'язні компоненти орграфа G = (V, E) через два пошуки в глибину: в графі G та графі G^T.
- Час роботи алгоритму Θ(V+E).

Сильно зв’язні компоненти

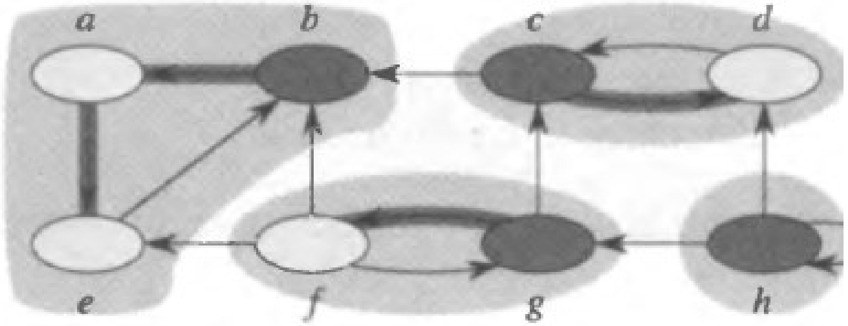
- При розгляді вершин при другому пошуку (в G^T) по суті відвідуються вершини графа компонентів в порядку топологічного сортування.
- Вибирається деяка вершина, що належить сильно зв’язній компоненті з максимальним часом завершення. З неї можна відвідати всі вершини лише цієї компоненти.
- Далі аналогічно вибирається вершина з наступної за часом завершення сильно зв’язної компоненти, робиться обхід і т. д.
- Таким чином, кожне дерево пошуку в глибину буде рівно однією сильно зв’язною компонентою.

Сильно зв’язні компоненти

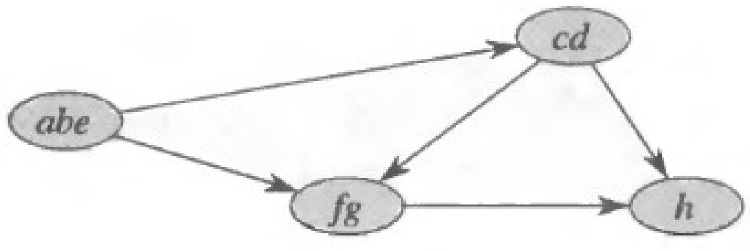
(а)



(б)



(в)



Графи G, G^T з пошуком в глибину на них і ациклічний граф сильно зв’язних компонентів.