Алгоритми та складність

II семестр

Лекція 9

Поняття динамічного програмування

- «Програмування» означає «планування».
- Часто згадується як табличний метод.
- Підхід, що дозволяє розв’язувати задачі, комбінуючи розв’язання допоміжних підзадач.
- В методі «розділяй та владарюй» задача розбивається на рекурсивні підзадачі, з розв’язку яких формується розв’язок вихідної задачі.
- Якщо підзадачі перекриваються, підхід «розділяй та владарюй» багатократно має розв’язувати ті самі підзадачі.
- Алгоритми динамічного програмування розв’язують кожну задачу один раз, записуючи результат в таблицю, що дозволяє не робити повторних обчислень.

Поняття динамічного програмування

- Метод – приклад *просторово-часового компромісу*, тобто використовується додаткова пам'ять для пришвидшення обчислень.
- За певних умов можна перетворити експоненціальний час роботи на поліноміальний.
- Як правило, динамічне програмування застосовується до *задач оптимізації*: задача може мати багато розв'язків, з якими пов'язані певні значення; серед усіх варіантів треба вибрати той, значення якого оптимальне (максимальне чи мінімальне).
- Умова застосовності динамічного програмування: задача повинна мати *оптимальну підструктуру*: оптимальний розв'язок задачі включає оптимальні розв'язки підзадач, що можуть бути незалежно розв'язані.

Поняття динамічного програмування

- Існує два еквівалентних способи реалізації підходу динамічного програмування.

Низхідний з запам’ятовуванням.

- Процедура пишеться рекурсивно, але вона модифікується таким чином, щоб розв’язок кожної підзадачі запам’ятовувася (зазвичай в масиві чи хеш-таблиці).
- В першу чергу перевіряється, чи вже була розв’язана підзадача, і або одразу повертається результат, або звичним чином виконуються обчислення.
- Про таку рекурсивну процедуру кажуть, що вона *з запам’ятовуванням*.

Поняття динамічного програмування

Висхідний.

- Зазвичай є залежність від певного природного поняття розміру підзадачі така, що розв’язок конкретної підзадачі залежить тільки від розв’язку менших підзадач.
- Підзадачі сортуються за розміром за зростанням.
- Розв’язуючи якусь підзадачу, треба розв’язати всі менші підзадачі, від яких вона залежить, і зберегти отримані розв’язки.
- Кожна підзадача розв’язується лише один раз, і в момент, коли доходимо до неї, всі необхідні для її розв’язання підзадачі вже отримали результат.
- Обидва підходи зазвичай приводять до алгоритмів з однаковим асимптотичним часом роботи, але висхідний варіант частіше дає кращі константи.

Жадібні алгоритми

- Часом розв'язання задач оптимізації не потребує розгляду і розв'язку всіх підзадач.
- Жадібний підхід передбачає побудову розв'язку, при якій на кожному кроці отримується частковий розв'язок початкової задачі, поки не отримається повний. При цьому на кожному кроці вибір має бути
  – допустимим – задовольняти обмеження задачі;
  – локально оптимальним – найкращим серед допустимих варіантів на цьому кроці;
  – остаточним – не може бути зміненим на наступних кроках алгоритму.
- Для низки задач такий постійний локально оптимальний вибір врешті-решт приводить оптимального розв'язку глобальної задачі.

Порівняння жадібного підходу та динамічного програмування

- Наявність оптимальної підструктури в задачі необхідна як для динамічного програмування, так і в жадібному підході. В чому ж буде різниця?
- Вибір, що робиться на кожному етапі в динамічному програмуванні, зазвичай залежить від розв'язків підзадач.
- Найтиповішим є висхідний напрям розв'язання, коли спочатку розв'язуються менші задачі, а потім – більші. Навіть при низхідному русі з запам'ятовуванням залежні задачі вже мають бути розв'язаними. Тому в будь-якому випадку *підзадачі розв'язуються до здійснення вибору*.
- Жадібний підхід використовує низхідну стратегію, при цьому *вибір завжди робиться до розв'язання підзадач*.

Порівняння жадібного підходу та динамічного програмування

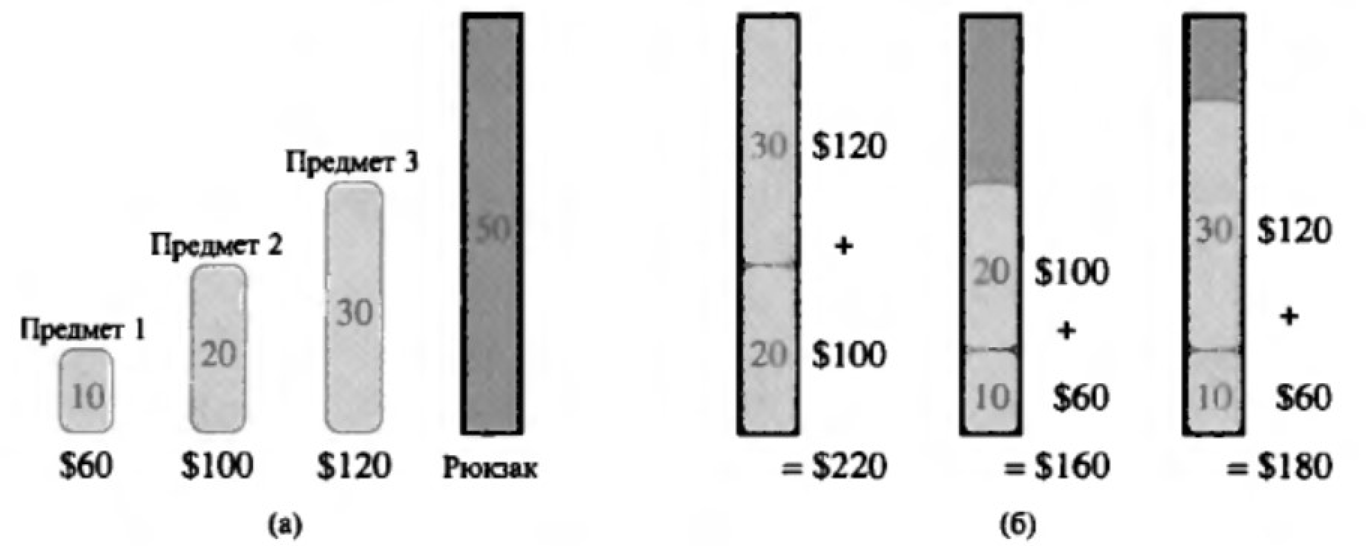
- Розглянемо два варіанти задачі про рюкзак.
- Дискретна задача про рюкзак. Є n предметів, і-й предмет має ціну v_i та вагу w_i (цілочисельні). Потрібно вибрати предмети найбільшої сумарної вартості, за умови цілочисельного обмеження загальної ваги W. Кожен предмет можна взяти лише один раз і цілим.
- Континуальна задача про рюкзак дозволяє брати частину предмета.
- Можливий контекст – злодій та золоті злитки (дискретна задача) чи золотий пісок (континуальна задача).

Порівняння жадібного підходу та динамічного програмування

- Обидві задачі мають властивість оптимальної підструктури.
- Якщо витягти з рюкзака предмет, то решта предметів мають бути найціннішими, якщо не враховувати цей предмет (чи його частину) і зменшити W на його вагу.
- Але лише континуальна задача дозволяє жадібну стратегію!
- Обчислюється питома вартість одиниці кожного товару і завантажується якомога більше товару з максимальною питомою вартістю. Якщо набрана вага менша за допустиму, аналогічно вибирається товар з максимальною питомою вартістю серед тих, що залишилися і т.д.

Порівняння жадібного підходу та динамічного програмування

- Для дискретної задачі жадібний підхід не спрацює.



- Незважаючи на те, що питома вартість першого предмета найвища, оптимальний розв’язок не містить його взагалі: потрібно взяти другий і третій предмети.

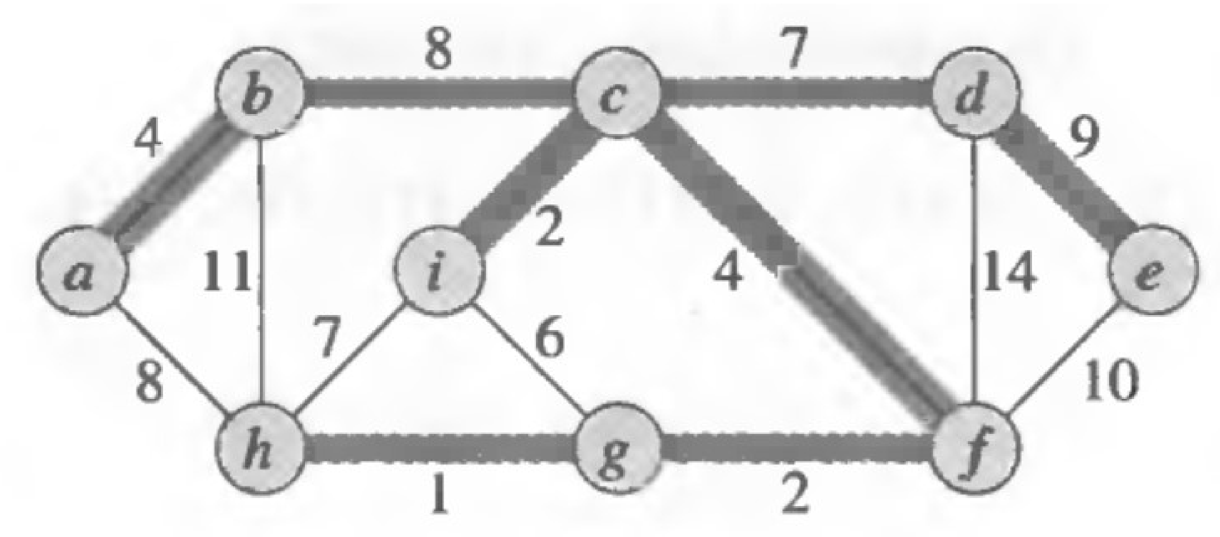
Мінімальні кістякові дерева

- В багатьох ситуацій природним чином виникає задача: з’єднати *n* точок так, щоб існував шлях між будь-якою парою точок, причому сумарна вартість з’єднань має бути мінімальною.
- Наприклад, з’єднати *n* контактів в електронній схемі, використавши мінімальну кількість дроту.
- Іншими словами, маючи зв’язний неорієнтований зважений граф G = (V, E), треба знайти ациклічну підмножину T $\subseteq$ E, яка з’єднує всі вершини та має мінімальну сумарну вагу:

$$w(T) = \sum_{(u,v)\in T} w(u,v)$$

- Утворена множина T – *кістякове дерево*, а задача його пошуку – *задача пошуку мінімального кістякового дерева*.

Мінімальні кістякові дерева



Мінімальне кістякове дерево зв’язного графа

- Біля ребер вказана їх вага.
- Виділені ребра мінімального кістякового дерева.
- Вказане дерево не є єдиним мінімальним: замінивши ребро (*b*,*c*) на (*a*,*h*), отримаємо інше кістякове дерево з такою самою вагою 37.

Мінімальні кістякові дерева

- Розглянемо узагальнений метод побудови мінімального кістякового дерева, який нарощує поточне кістякове дерево по одному ребру.
- Використаємо жадібну стратегію: на кожному кроці вибиратимемо найкращий з поточних можливих варіантів.
- Маємо зв'язний неорієнтований граф G = (V, E) з дійсною ваговою функцією *w*.
- Працюємо з множиною ребер A, підтримуючи наступний інваріант циклу:

перед кожною черговою ітерацією A є підмножиною деякого мінімального кістякового дерева графа G.

Мінімальні кістякові дерева

- На кожному кроці алгоритму визначається ребро (*u*,*v*), яке можна додати до A без порушення інваріанту: $A \cup \{(u,v)\}$ також має бути підмножиною мінімального кістякового дерева.
- Назвемо таке ребро (*u*,*v*) *безпечним* для A: його можна додати до A, не порушивши інваріант.
- Схема алгоритму наступна:

```
GENERIC-MST(G, w)
A = ∅
while A не образует остовного дерева
    Найти ребро (u, v), безопасное для A
    A = A ∪ {(u, v)}
return A
```

Мінімальні кістякові дерева

- Алгоритм працюватиме коректно – доведемо це.

Ініціалізація. Після рядка 1 множина А тривіально задовольняє інваріант.

Збереження. Цикл зберігає інваріант, оскільки додає тільки безпечні ребра.

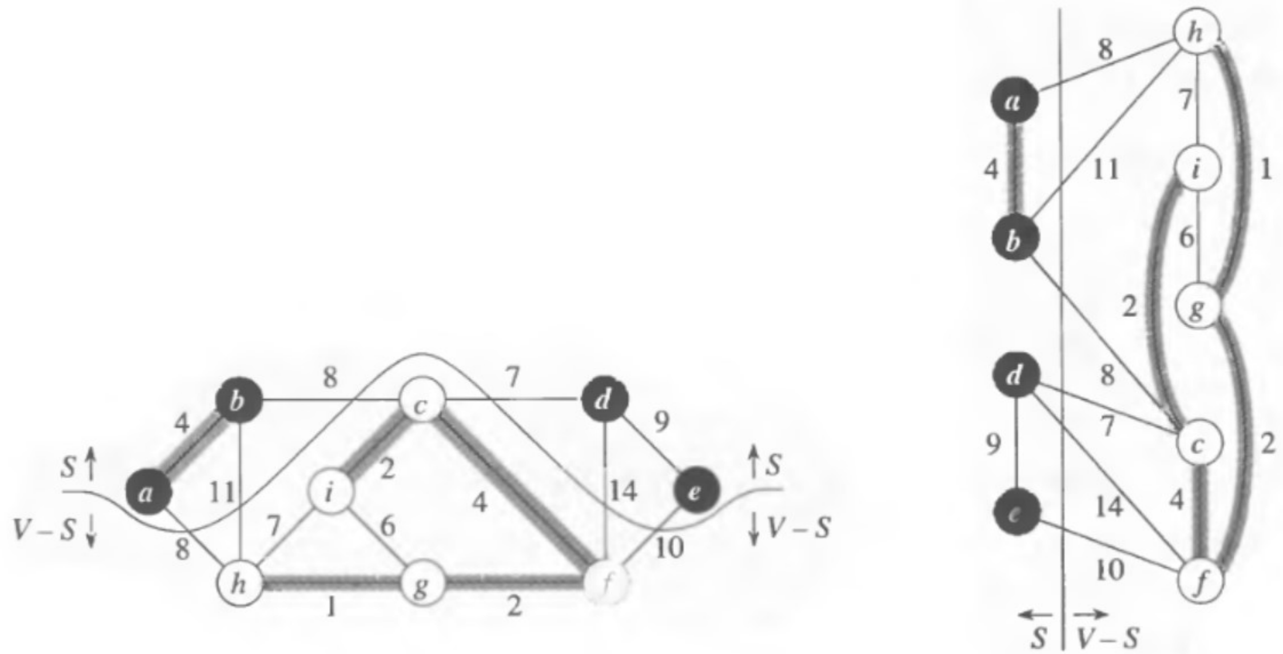
Завершення. Всі ребра, додані до А, входять до мінімального кістякового дерева, тому множина А, яку поверне алгоритм, буде мінімальним кістяковим деревом.

- Залишається питання, яким чином можна розпізнавати і ефективно знаходити безпечні ребра.

Мінімальні кістякові дерева

- *Розріз* {S, V–S} неорієнтованого графа G = (V, E) – деяке розбиття V.
- Ребро (u,v)$\in$E *перетинає* розріз {S, V–S}, якщо один його кінець належить до множини S, а інший до V–S.
- Розріз *узгоджений* з множиною ребер A, якщо жодне ребро з A не перетинає розріз.
- Ребро, що перетинає розріз, називається *легким*, якщо воно має мінімальну вагу серед усіх ребер, які перетинають розріз. Може існувати декілька легких ребер одночасно.
- В загальному випадку ребро називають *легким ребром*, що задовольняє деяку умову, якщо воно має мінімальну вагу серед усіх ребер, що задовольняють цю умову.

Мінімальні кістякові дерева



Два варіанти представлення розрізу {S, V–S}

- Чорні вершини належать до множини S, а білі – до V–S. Ребра, що перетинають розріз, з’єднують пари різнокольорових вершин.
- (*d*,*c*) – єдине легке ребро, що перетинає розріз.
- Підмножина ребер A заштрихована, причому розріз {S, V–S} узгоджений з A – жодне ребро з A його не перетинає.

Мінімальні кістякові дерева

Теорема. Нехай G = (V, E) – зв'язний неорієнтований граф з дійсною ваговою функцією *w*, що визначена на E. Нехай A – підмножина E, яка включається до деякого мінімального кістякового дерева G, маємо {S, V–S} – довільний узгоджений з A розріз G та (*u*,*v*) – легке ребро, що перетинає {S, V–S}. Тоді ребро (*u*,*v*) є безпечним для A.

Наслідок. Нехай G = (V, E) – зв'язний неорієнтований граф з дійсною ваговою функцією *w*, що визначена на E. Нехай A – підмножина E, яка включається до деякого мінімального кістякового дерева G та $C=(V_C,E_C)$ – зв'язна компонента (дерево) в лісі $G_A=(V,A)$. Якщо (*u*,*v*) – легке ребро, що з'єднує C з деякою іншою компонентою в G_A, то ребро (*u*,*v*) є безпечним для A.

Мінімальні кістякові дерева

- В процесі роботи алгоритму множина A завжди ациклічна.
- В будь-який момент виконання алгоритму граф $G_A=(V,A)$ є лісом, а кожна з його зв'язних компонент – деревом (в тому числі з однієї вершини).
- Кожне безпечне для A ребро (u,v) з'єднує різні компоненти G_A (бо множина $A \cup \{(u,v)\}$ має бути ациклічною).
- Цикл виконується |V|–1 разів: він знаходить по одному з |V|–1 ребер мінімального кістякового дерева при кожній ітерації.
- Спочатку $A=\varnothing$ та G_A містить |V| дерев. Кожна ітерація зменшує їх кількість на 1. Алгоритм завершується, коли ліс складатиметься з одного дерева.

Мінімальні кістякові дерева

- Наслідок теореми використовується алгоритмами Прима та Крускала.
- Кожен з них використовує своє правило для визначення безпечних ребер.
- В алгоритмі *Крускала* множина А є лісом, куди додаються безпечні ребра – ребра мінімальної ваги, що з'єднують *дві різні компоненти*.
- В алгоритмі *Прима* множина А утворює єдине дерево, в яке додаються безпечні ребра – ребра мінімальної ваги, що з'єднують дерево *з вершиною поза деревом*.

Алгоритм Крускала

- Джозеф Крускал (Joseph Kruskal) відкрив цей алгоритм навчаючись на другому курсі.
- Алгоритм вибирає безпечне ребро для додавання до лісу, шукаючи ребро (u,v) з мінімальною вагою серед усіх ребер, що з’єднують два дерева в лісі.
- Позначимо дерева, які з’єднує ребро (u,v), через C_1 та C_2.
- Оскільки (u,v) має бути легким ребром, що з’єднує C_1 з деяким іншим деревом, з наслідку теореми отримуємо: (u,v) – безпечне для C_1 ребро.
- Алгоритм Крускала є дійсно жадібним, бо на кожному кроці додає до лісу ребро мінімально можливої ваги.

Алгоритм Крускала

- Алгоритм насамперед сортує ребра за неспаданням їх ваг.
- Множина A ініціалізується як порожня і створюються |V| дерев з однієї вершини.
- Відсортовані ребра переглядаються, починаючи з найлегшого. Перевіряється, чи належать кінці (u,v) різним деревам.
- Якщо так, ребро (u,v) додається до множини A і вершини двох відповідних дерев об’єднуються.
- Інакше ребро належить одному дереву і не може бути доданим до лісу без утворення циклу, а тому воно відкидається.

Алгоритм Крускала

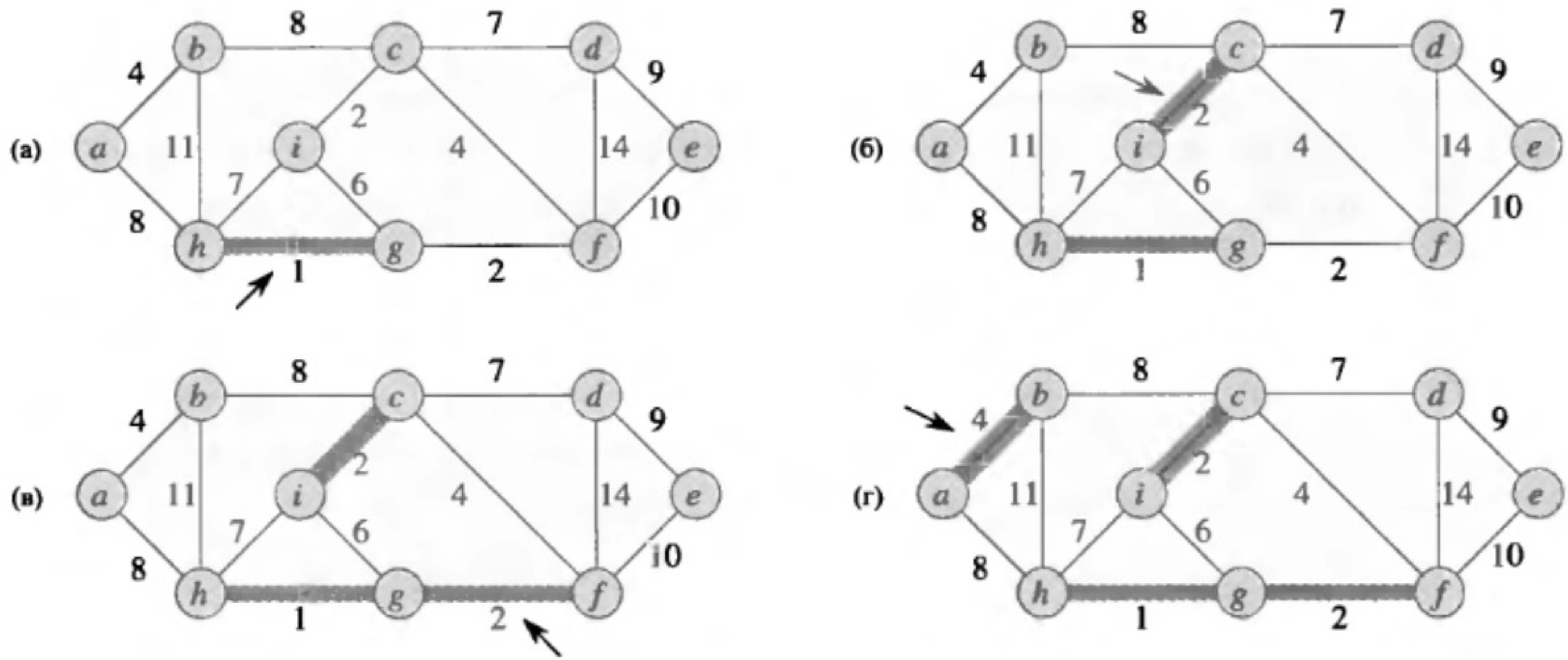
```
MST-KRUSKAL(G, w)
A = ∅
for каждой вершины v ∈ G.V
    MAKE-SET(v)
Отсортировать ребра G.E в неуменьшающемся порядке по весу w
for каждого ребра (u, v) ∈ G.E в этом порядке
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
return A
```

- Реалізація алгоритму використовує структуру для представлення множин, що не перетинаються.
- Кожна множина містить вершини деякого дерева в поточному лісі.
- Операція FIND-SET(*u*) повертає представника множини, що містить *u*. Отже, для перевірки, чи належать вершини *u* та *v* одному дереву, треба перевірити рівність FIND-SET(*u*) та FIND-SET(*v*).
- Операція UNION(*u*,*v*) об’єднує дерева *u* та *v*.

Алгоритм Крускала

- Час роботи алгоритму Крускала залежить від конкретної реалізації структури даних для множин, що не перетинаються.
- Якщо ліс множин, що не перетинаються, реалізований з урахуванням евристик об'єднання за рангом та стиснення шляху (найшвидша відома реалізація), часова оцінка алгоритму Крускала визначатиметься часом сортування ребер O(E log E).
- Слід зауважити, що $|E|<|V|^2$, тому $\log(|E|)=O(\log V)$.
- Тому час роботи алгоритму Крускала іноді записують як O(E log V).

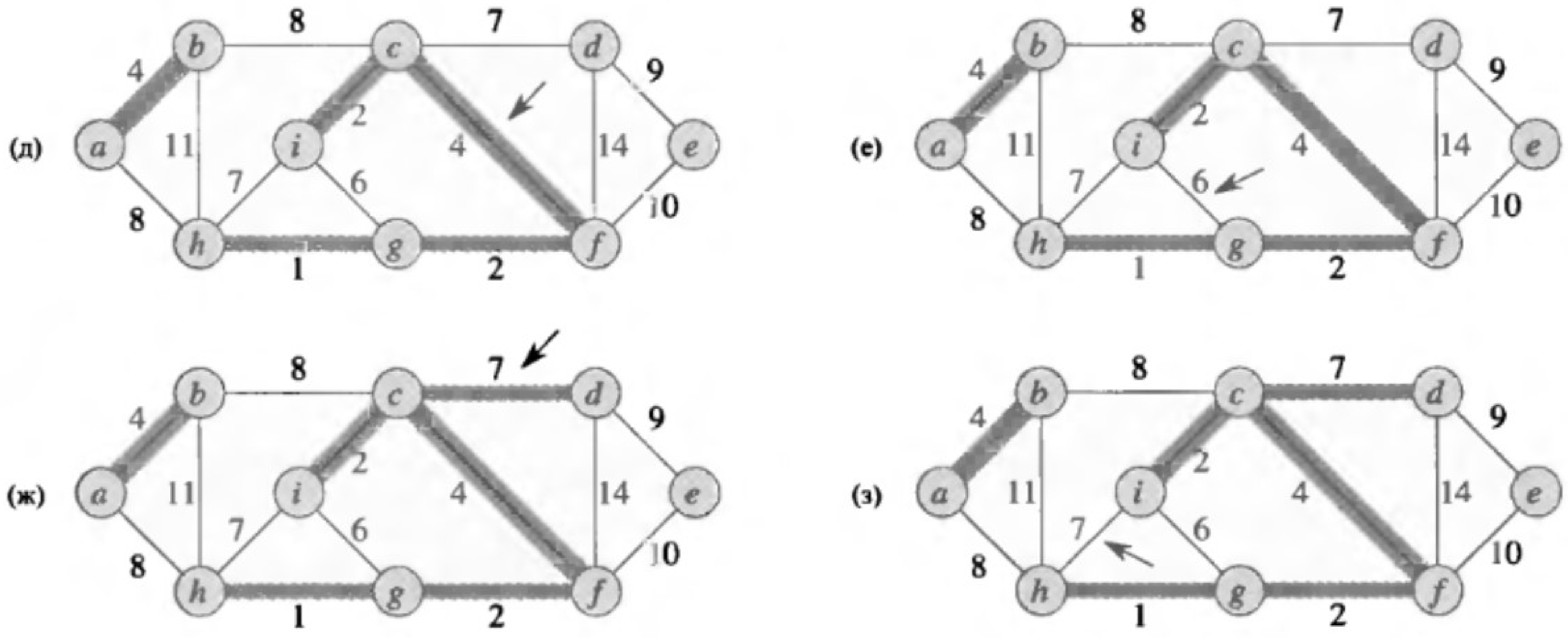
Алгоритм Крускала



Приклад роботи алгоритму Крускала (1)

- Затемнені ребра належать лісу, що зростає.
- Стрілка показує на чергове ребро за зростанням ваги, що розглядається.

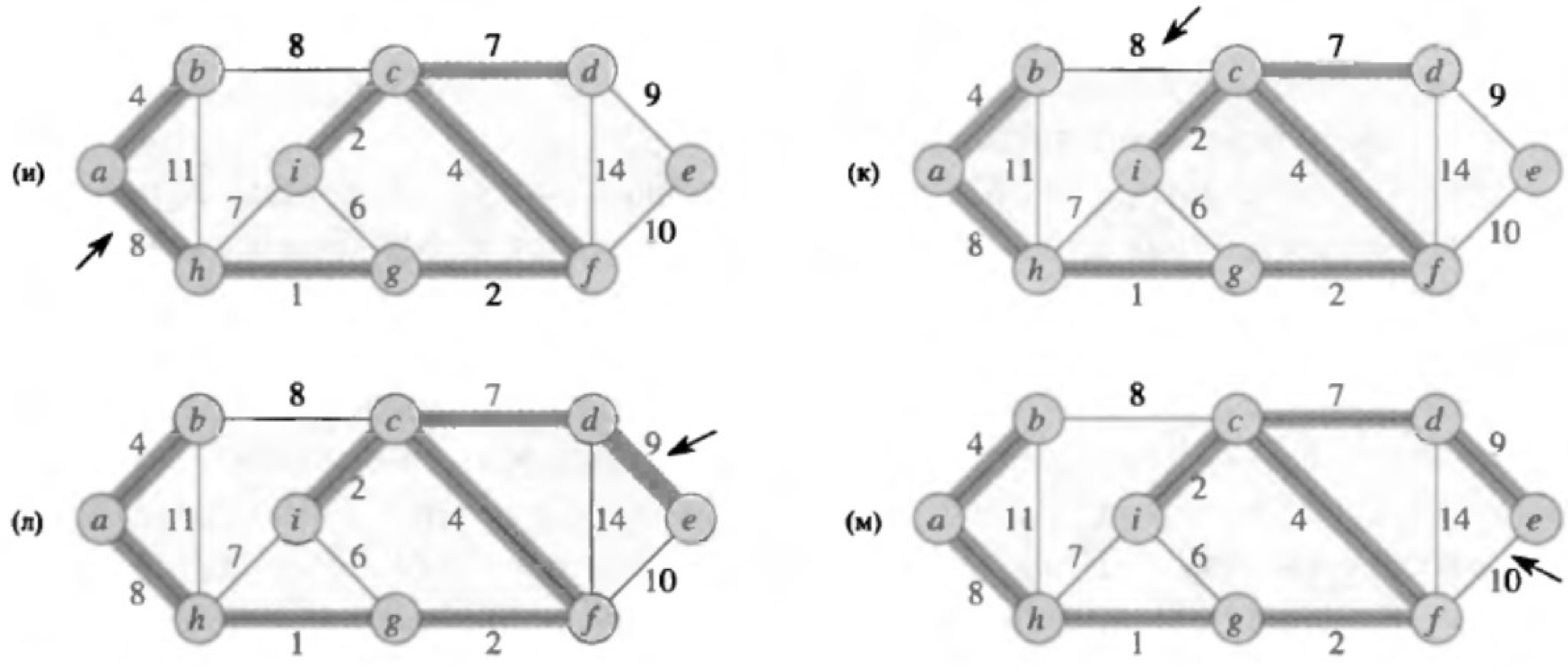
Алгоритм Крускала



Приклад роботи алгоритму Крускала (2)

- Затемнені ребра належать лісу, що зростає.
- Стрілка показує на чергове ребро за зростанням ваги, що розглядається.

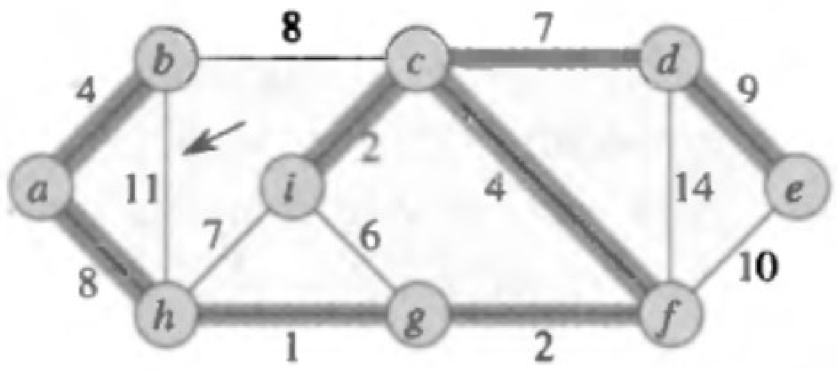
Алгоритм Крускала

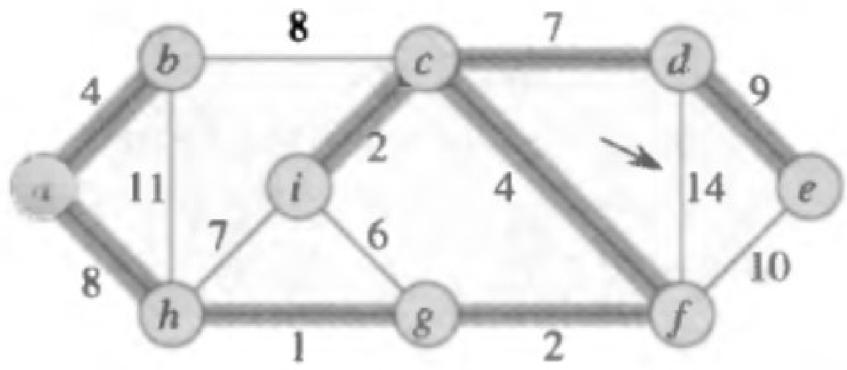


Приклад роботи алгоритму Крускала (3)

- Затемнені ребра належать лісу, що зростає.
- Стрілка показує на чергове ребро за зростанням ваги, що розглядається.

Алгоритм Крускала

(н)


(о)


Приклад роботи алгоритму Крускала (4)

- Затемнені ребра належать лісу, що зростає.
- Стрілка показує на чергове ребро за зростанням ваги, що розглядається.

Робота з множинами, що не перетинаються

- *Структура даних для множин, що не перетинаються*, підтримує набір множин $S=\{S_1,\ldots,S_k\}$, що не перетинаються.
- Кожна множина ідентифікується *представником* – деяким елементом множини.
- Важливо, щоб при повторних запитах представника множини вибирався той самий елемент (за умови відсутності змін в множині між запитами).
- Іноді вимагається, щоб вибирався конкретний елемент множини (наприклад, найменший).
- Зазвичай вважають, що елементи множини є цілими числами чи можуть бути відображені на **Z**.

Робота з множинами, що не перетинаються

- Потрібно забезпечити підтримку наступних операцій.

MAKE-SET(*x*) створює нову множину з єдиного члена-представника *x*. При цьому *x* не може належати іншій множині (умова неперетину множин).

UNION(*x*,*y*) об'єднує динамічні множини, що містять *x* та *y*. За умовою, вони не мають перетинатися. Вихідні множини при цьому знищуються. Представником отриманого об'єднання може обиратися довільний його елемент.

FIND-SET(*x*) повертає представника (єдиної) множини, що містить елемент *x*.

- Аналізується як час роботи *n* операцій MAKE-SET, так і послідовності *m* операцій всіх трьох типів.

Робота з множинами, що не перетинаються

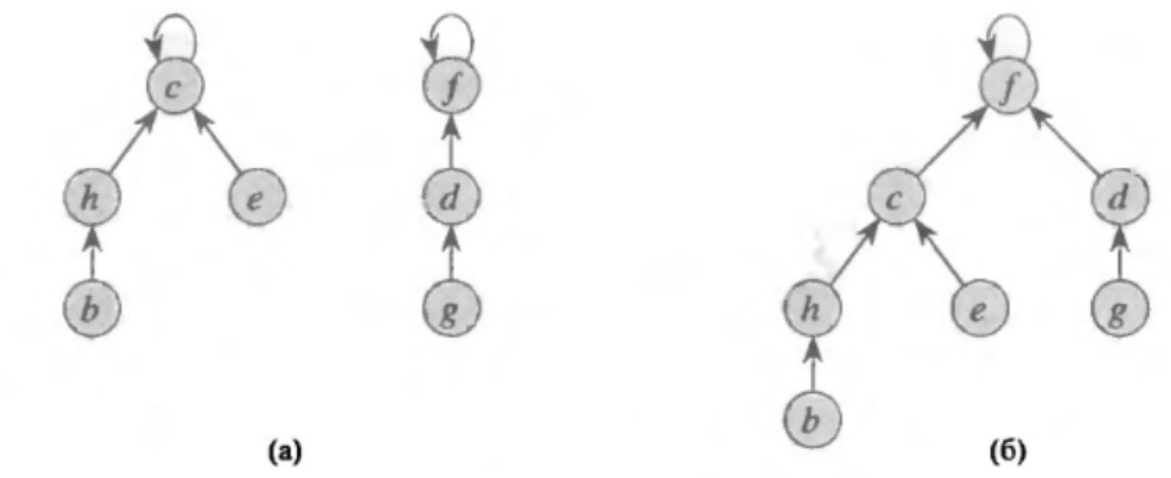
- Існує два альтернативних підходи до реалізації. Перший (*швидкий пошук*) оптимізує часову ефективність пошуку, другий (*швидке об'єднання*) – об'єднання.
- Підхід *швидкого пошуку* використовує зв'язані списки: кожна множина представлена своїм списком.
- Представником є перший елемент списку.
- Кожна з операцій MAKE-SET та FIND-SET виконується за O(1); амортизований час для n операцій UNION складає $\Theta(n)$.
- При використанні *вагової евристики* (вводиться поле довжини списку, коротший список завжди додається до довшого) послідовність з m операцій всіх трьох типів, з яких n операцій MAKE-SET, виконується за час $O(m + n \log n)$.

Робота з множинами, що не перетинаються

• Підхід *швидкого об’єднання* ефективніший, множини представляються кореневими деревами.

• Представником є корінь дерева.

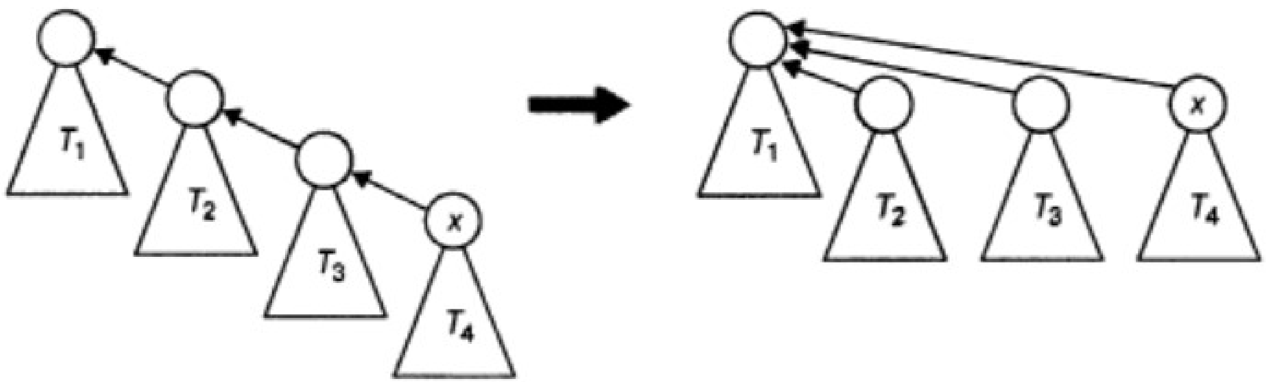
• Ребра направлені від дочірніх вузлів до батьківських.



• Операція UNION підв’язує корінь одного дерева до іншого.

Робота з множинами, що не перетинаються

- *Евристика об'єднання за рангом* аналогічна ваговій евристиці при списковому представленні: «менше» дерево прив'язується до «більшого».
- Замість явного розміру вводиться поняття *рангу* кореня – верхньої границі висоти вузла. При виконанні UNION корінь з меншим рангом має вказувати на корінь з більшим рангом.
- *Евристика стиснення шляху*: кожен вузол, що зустрівся в процесі операції FIND-SET, перенаправляється на корінь:



Робота з множинами, що не перетинаються

- Нехай є послідовність з m операцій всіх трьох типів, з яких n операцій MAKE-SET.
- Використання евристики об’єднання за рангом дає час роботи $O(m \log n)$.
- Використання обох евристик одразу дасть часову оцінку $O(m \cdot \alpha(n))$.
- Тут $\alpha(n)$ – дуже, дуже повільно зростаюча функція (обернена до функції Аккермана).
- Для всіх мислимих практичних застосувань при роботі з множинами, що не перетинаються, $\alpha(n) \leq 4$.
- Таким чином, можна розглядати час роботи на практиці як фактично лінійний.

Алгоритм Прима

- Алгоритм використовує той факт, що ребра в множині A завжди мають утворити єдине дерево.
- Побудова дерева розпочинається з довільної кореневої вершини. Дерево зростає, поки не охопить всі вершини у V.
- На кожному кроці до дерева A додається легке ребро, що з'єднує дерево з деякою вершиною із залишку графа.
- Згідно наслідку теореми, за таким правилом додаються лише безпечні для A ребра. Отже, в результаті з ребер A отримуємо мінімальне кістякове дерево.
- Алгоритм Прима є дійсно жадібним, бо на кожному кроці додає до дерева ребро, яке вносить мінімально можливий вклад до сумарної ваги.

Алгоритм Прима

- Для ефективної реалізації алгоритму треба вміти швидко вибирати нове ребро для додавання до дерева.
- На початку потрібно задати корінь *r*, з якого виросте мінімальне кістякове дерево.
- В процесі роботи всі вершини, що не належать дереву, заносяться до неспадаючої черги з пріоритетами Q за атрибутом *key*.
- Для кожної вершини *v* значення *v.key* позначає мінімальну вагу серед всіх ребер, що з'єднують *v* з вершиною дерева. Якщо такого ребра немає, покладемо *v.key* = ∞.
- Атрибут $v.\pi$ вказує на предка *v* в дереві.

Алгоритм Прима

- На початку ключі *key* всіх вершин, крім кореня, встановлюються як ∞. Щоб корінь обробився першим, покладають $r.key = 0$.
- Для всіх вершин предки встановлюються як NIL.
- Множина А неявно підтримується як

$$A=\{(v,v.\pi): v\in V - \{r\} - Q\}.$$

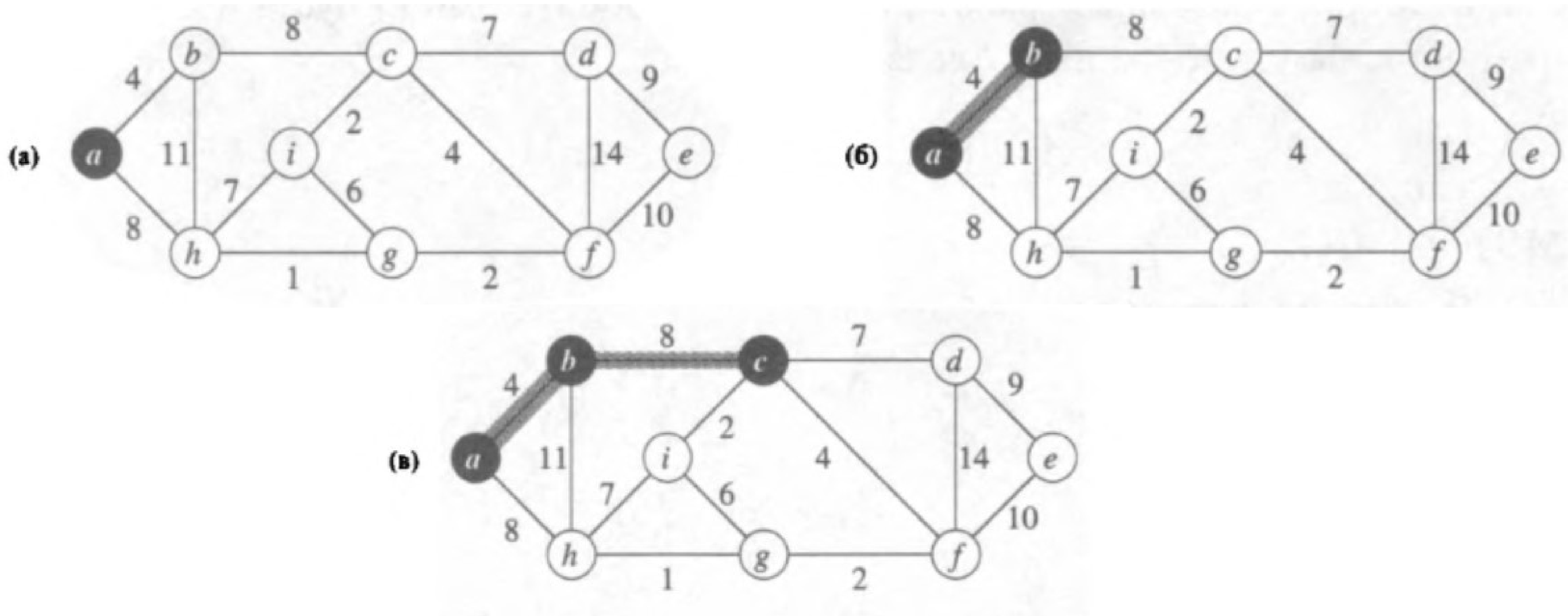
- Всі вершини заносяться до черги з пріоритетами.
- На кожній ітерації витягається вершина $u\in Q$, інцидентна легкому ребру, що перетинає розріз $\{V, V–Q\}$.
- Видалення *u* з Q додає її до множини V–Q вершин дерева, одночасно додаючи $(u,u.\pi)$ до А.
- Далі потрібно оновити атрибути *key* та π для всіх вершин, що не належать до дерева та суміжних з *u*.
- В кінці роботи алгоритму черга з пріоритетами порожня і мінімальним кістяковим деревом для G буде дерево $A=\{(v,v.\pi): v\in V - \{r\}\}$.

Алгоритм Прима

```
MST-PRIM(G, w, r)
for каждой вершины u ∈ G.V
    u.key = ∞
    u.π = NIL
r.key = 0
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    for каждой вершины v ∈ G.Adj[u]
        if v ∈ Q и w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
            // С вызовом DECREASE-KEY(Q, v, w(u, v))
```

- Час роботи алгоритму Прима залежить від реалізації черги з пріоритетами Q.
- Якщо використати використання бінарну піраміду, він складе O(E logV).
- За умови використання пірамід Фібоначчі час покращиться до O(E +V log V).

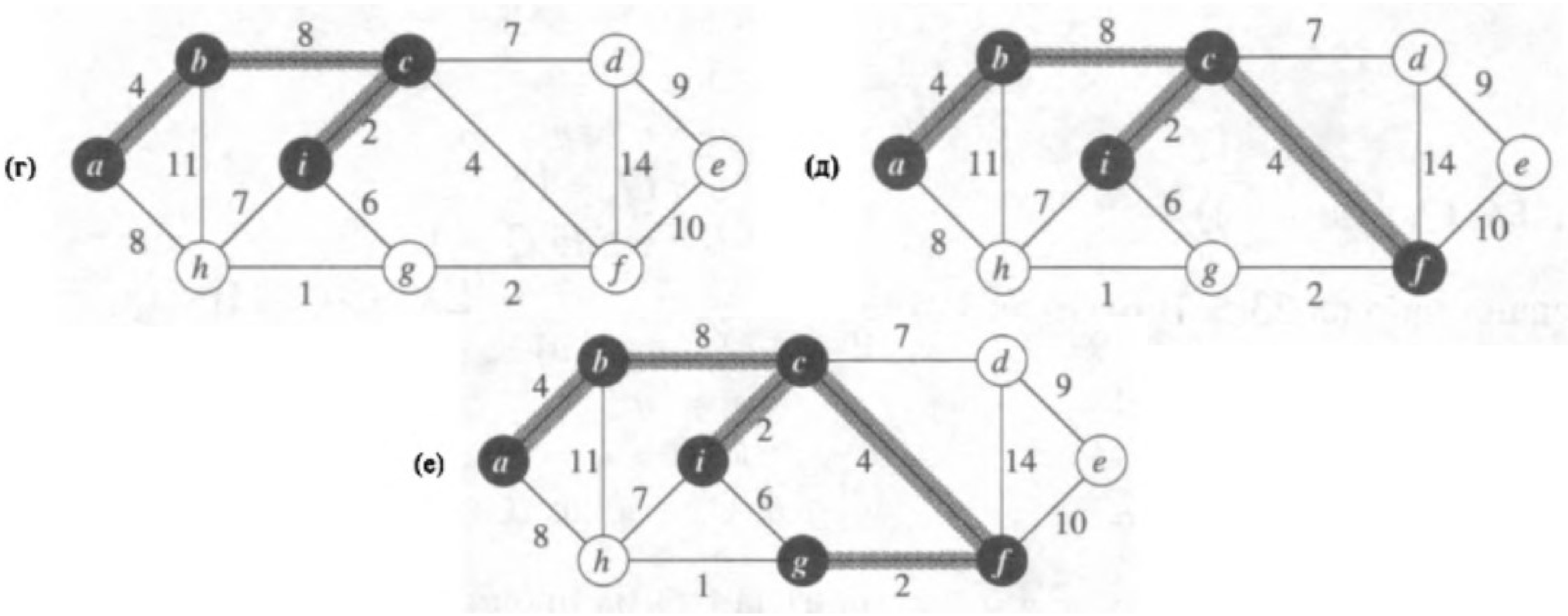
Алгоритм Прима



Приклад роботи алгоритму Прима (1)

- Кореневою вершиною є *a*.
- Затемнені ребра належать дереву, що зростає; його вершини – чорні.
- На кожній ітерації вершини дерева визначають розріз графа, і до дерева додається легке ребро, яке перетинає розріз.

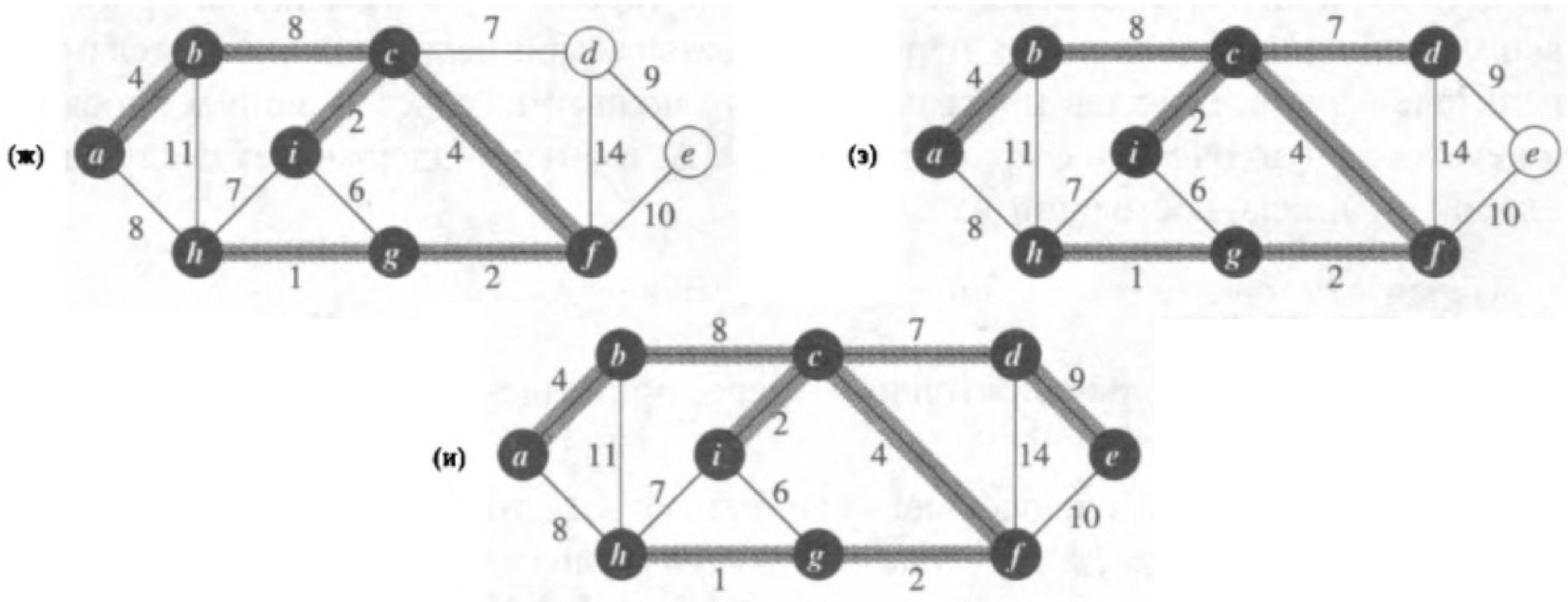
Алгоритм Прима



Приклад роботи алгоритму Прима (2)

- Кореневою вершиною є *a*.
- Затемнені ребра належать дереву, що зростає; його вершини – чорні.
- На кожній ітерації вершини дерева визначають розріз графа, і до дерева додається легке ребро, яке перетинає розріз.

Алгоритм Прима



Приклад роботи алгоритму Прима (3)

- Кореневою вершиною є *a*.
- Затемнені ребра належать дереву, що зростає; його вершини – чорні.
- На кожній ітерації вершини дерева визначають розріз графа, і до дерева додається легке ребро, яке перетинає розріз.