Київський національний університет імені Тараса Шевченка
факультет комп'ютерних наук та кібернетики

ОСНОВИ КОМП'ЮТЕРНИХ АЛГОРИТМІВ.

ЛЕКЦІЇ

ТВОРИ
2021

УДК 004.43.421(042.3)

Рекомендовано вченою радою факультету комп’ютерних наук та кібернетики Київського національного університету імені Тараса Шевченка (протокол №15 від 7 червня 2021 р.).

Укладач: кандидат фіз.-мат. наук, професор, доктор габілітації Вергунова І.М.

Рецензенти: доктор фіз.-мат. наук, доцент І.О. Завадський
доктор наук, професор В.Р. Стебловська

В 52 Вергунова І.М. Основи комп'ютерних алгоритмів. Лекції. – Вінниця : ТВОРИ, 2021. – 228 с.

ISBN 978-966-949-924-0

Викладено лекційний курс з дисципліни «Основи комп'ютерних алгоритмів» для студентів факультету комп’ютерних наук та кібернетики спеціальності 122 «Комп'ютерні науки» ОП «Інформатика».

УДК 004.43.421(042.3)
ISBN 978-966-949-924-0

Лекція 1. Хешування та некриптографічні алгоритми хешування

Хеш-таблиця – це структура даних, що дозволяє ефективно реалізовувати словники (часто середній час пошуку елементу $O(1)$). Узагальнює поняття масиву. Наявність прямої індексації елементів масиву забезпечує доступ до довільної позиції у масиві за час $O(1)$. В хеш-таблицях також може бути пряма індексація (якщо можемо виділити масив такого розміру, щоб кожне можливе значення ключа мало свою комірку). Але звичайно кількість ключів, що зберігаються в масиві, менша у порівнянні з кількістю можливих значень ключів. В таких випадках використовують хеш-таблиці, що використовують масиви, розмір яких пропорційний кількості ключів, що реально там зберігаються. Замість безпосереднього використання ключа як індексу масиву індекс обчислюється за значенням ключа.

Таблиці з прямою адресацією. Використовують для невеликої сукупності ключів. Нехай маємо справу із динамічною множиною, кожен елемент якої має ключа з $U = \{0,1,..., m-1\}$, де m є не дуже великим та жодні два елементи не мають однакових ключів. Для представлення динамічної множини використаємо таблицю з прямою адресацією (масив) $T[0..m-1]$, кожна комірка якої відповідає ключу із сукупності $U = \{0,1,..., m-1\}$. Тоді комірка k вказує на елемент множини з ключом k. Якщо множина не містить елемента з таким ключем, то $T[k] = NIL$. Тоді виконання словникових операцій (пошук, вставка, вилучення) відбувається за час $O(1)$ (рис. 1.1). Іноді множина реальних ключів визначає комірки в таблиці, які містять вказівники на елементи динамічної множини (рис.1.1), а іноді елементи можуть зберігатися безпосередньо в таблиці з прямою адресацією (в комірках, що дозволяє економити пам'ять за рахунок відмови від зберігання ключів та супутніх даних елементів в об’єктах, зовнішніх по відношенню до таблиці, та від вказівників на ці об’єкти у таблиці, але для вказівки пустої комірки користуються спеціальним значенням ключа). Якщо елементи можуть зберігатися безпосередньо в таблиці з прямою адресацією, то часто збереження ключа не є необхідним, так як знаючи індекс об’єкта у таблиці будемо знати й ключ.

```
    DIRECT-ADDRESS-SEARCH(T, k)
1   return T[k]
    DIRECT-ADDRESS-INSERT(T, x)
1   T[x.key] = x
    DIRECT-ADDRESS-DELETE(T, x)
1   T[x.key] = NIL
```

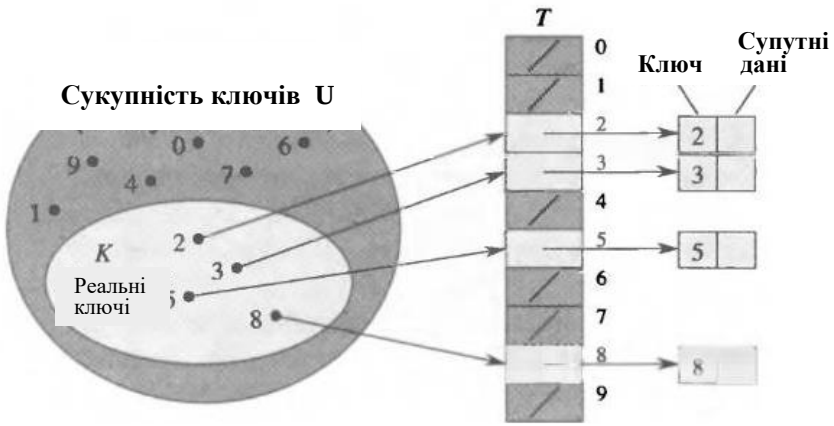


Рис. 1.1. Використання таблиці з прямою адресацією [1].

Але використання таблиці з прямою адресацією має явний недолік: для вже доволі великої сукупності ключів U зберігання таблиці T буде невигідним.

Використання *хеш-таблиці* дозволяє зменшити вимого до об'єму пам'яті до $\Theta(|K|)$ та зберегти час пошуку елементу як $O(1)$ *у середньому випадку*. Для випадку *прямої адресації* елемент з ключом k зберігається у комірці k, при використанні *хешування* - у комірці $h(k)$ (за рахунок використання **хеш-функції** h, що обчислює комірку для заданого ключа k).

Хеш-функція – це така функція $h: U \to \{0,1,..., m-1\}$, що відображає сукупність ключів U на комірки хеш-таблиці $T[0..m-1]$. Кажуть, що

елемент з ключем *k* хешується в комірку *h(k)*, а величину *h(k)* називають хеш-значенням ключа *k*.

Звичайно розмір хеш-таблиці значно менший $|U|$. Хеш-функція зменшує робочий діапазон індексів масиву і тому замість розміру $|U|$ значень обходимося масивом розміром *m* (рис. 1.2).

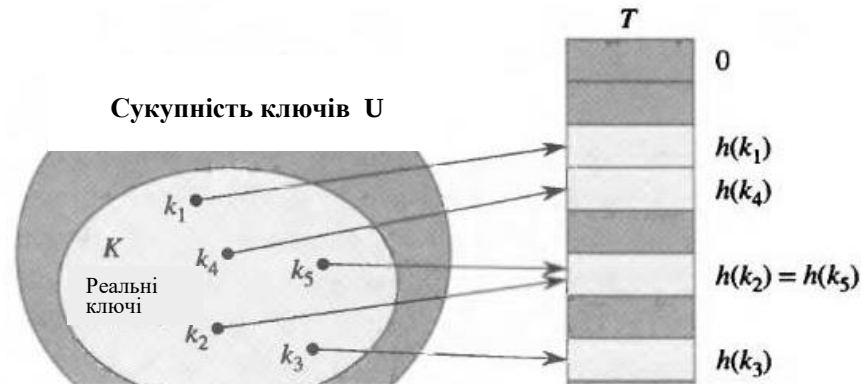


Рис. 1.2. Хешування до хеш-таблиці [1].

Ситуація, коли два ключа можуть бути хешовані в одну комірку називають **колізією**. Є різні технології вирішення колізій. За допомогою підходящого вибору хеш-функції намагаються їх мінімізувати. Але так як $|U| > m$, то повинно існувати не менше двох ключів, що мають однакове хеш-значення.

Найпростішим методом вирішення колізій є *метод ланцюгів*, за яким розташовують всі елементи, що хешовані до одної комірки, у зв'язаний (дво- або однозв'язний) список (рис. 1.3). Це *відкрите (зовнішнє) хешування*. Тоді, наприклад, комірка j містить вказівник на заголовок списку всіх елементів, хеш-значення яких дорівнює j, а якщо таких елементів немає, комірка містить значення NIL.

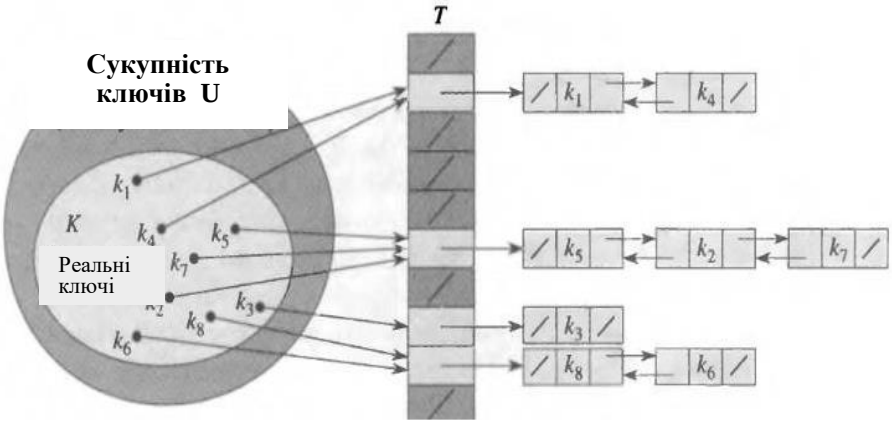


Рис. 1.3. Вирішення колізій за допомогою ланцюгів (відкрите, зовнішнє хешування) [1].

CHAINED-HASH-SEARCH(T, k)
1 Пошук елементу з ключом k у списку $T[h(k)]$
CHAINED-HASH-INSERT(T, x)
1 Вставка x до заголовку списку $T[h(x, key)]$
CHAINED-HASH-DELETE(T, x)
1 Вилучення x із списку $T[h(x, key)]$

Припустимо, що елемент, що вставляємо до таблиці, відсутній в ній. Тоді час, необхідний для вставки, у найгіршому випадку – $O(1)$. Інакше перевіряємо наявність, що тягне за собою додаткову вартість виконання пошуку елементу з відповідним ключом перед вставкою. Тоді час роботи буде пропорційним довжині списку.
Вилучення елементу матиме вартість $O(1)$ за умови використання двічі зв'язного списку.
Проведемо більш докладний *аналіз хешування з ланцюгами*. Припустимо, маємо хеш-таблицю T з m комірками, в яких зберігаються n елементів. *Коефіцієнтом заповнення таблиці* α (може бути менше, дорівнювати, більше 1) назовемо n/m.

У *найгіршому випадку* всі n ключів можуть бути хешовані в одну комірку. Тоді маємо список довжиною n. Тому час пошуку для найгіршого випадку буде $\Theta(n)$ + час обчислення хеш-функції, а тоді використання хеш-функції буде невигідне.

У *середньому випадку* продуктивність хешування залежить від того, як добре хеш-функція розподіляє множину ключів, що зберігається, за m комірками у середньому.

Розглянемо випадок **простого рівномірного хешування**. Тобто, вважаємо, що всі елементи хешуються у комірки рівномірно та незалежно. Якщо довжини списків позначимо $T[j]$ для $j = 0,1,\ldots, m\text{-}1$ як n_j, то $n = n_0 + n_1 + ... + n_{m-1}$, а очікуване значення n_j буде $E[n_j] = \alpha = n/m$.

У припущенні, що для обчислення хеш-значення $h(k)$ потрібно часу $O(1)$, час для пошуку елементу з ключом k, лінійно залежить від довжини $n_{h(k)}$ списку $T[h(k)]$. Не враховуючи час, що потрібен для обчислення хеш-функції, та для доступу до комірки $h(k)$, одержимо математичне очікування кількості елементів, що має бути перевірено алгоритмом пошуку (кількість елементів у списку $T[h(k)]$, які перевіряються на рівність їхніх ключів величині k).

1. Пошук невдалий та в таблиці немає елементів з ключом k.

Теорема. У хеш-таблиці з вирішенням колізій методом ланцюгів **час невдалого пошуку** у середньому випадку у припущенні простого рівномірного хешування складає $\Theta(1+\alpha)$ [1].

Доведення. За припущенням простого рівномірного хешування будь-який ключ k, який ще не знаходиться у таблиці, може бути розміщений з однаковою ймовірністю у будь-яку з m комірок. Математичне очікування часу невдалого пошуку ключа k дорівнює часу пошуку до кінця списку $T[h(k)]$ з очікуваною довжиною $E[n_{h(k)}] = \alpha$. Тому, за невдалого пошуку математичне очікування кількості елементів, що перевіряється, дорівнює α, а загальний час для пошуку, включаючи час обчислення хеш-функції $h(k)$, буде $\Theta(1+\alpha)$.

2. Пошук вдалий та в таблиці міститься елемент з ключом k.

З вдалого пошуку ймовірність пошуку у списку різна для різних списків та пропорційна кількості елементів, що в них міститься.

Теорема. У хеш-таблиці з вирішенням колізій методом ланцюгів **час вдалого пошуку** у середньому випадку у припущенні простого рівномірного хешування у середньому складає $\Theta(1+\alpha)$ [1].

Доведення. Нехай шуканий елемент з рівною ймовірністю може бути будь-яким елементом, що зберігається в таблиці. Кількість елементів, що перевіряється у процесі вдалого пошуку елементу *x*, буде на 1 більше, ніж кількість елементів, що знаходиться перед *x*. Елементи, що знаходяться у списку до *x*, були вставлені до списку після того, як елемент *x* був збережений у таблиці (нові елементи розміщаються на початку списку). Щоб знайти математичне очікування кількості елементів, що перевіряються, візьмемо середнє за всіма *n* елементами *x* в таблиці значення, яке дорівнює 1 + E{кількості елементів, доданих до списку *x* після самого шуканого елементу}. Нехай x_i – *i*-й елемент, вставлений до таблиці, $i = 1, 2, \ldots, n$, та $k_i = x_i.key$. Уведемо індикаторну випадкову величину $X_{ij} = I\{h(k_i) = h(k_j)\}$. У припущенні простого рівномірного хешування $\Pr\{h(k_i) = h(k_j)\} = 1/m$, а за основною властивістю індикаторної випадкової величини $E[X_{ij}] = 1/m$. Тоді математичне очікування кількості елементів, що перевіряються у випадку вдалого пошуку, буде:

$$E\left[\frac{1}{n}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}X_{ij})\right] = \frac{1}{n}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}E[X_{ij}]) = \frac{1}{n}\sum_{i=1}^{n}(1+\sum_{j=i+1}^{n}\frac{1}{m}) = 1+\frac{1}{nm}\sum_{i=1}^{n}(n-i) =$$

$$= 1+\frac{1}{nm}(\sum_{i=1}^{n}n-\sum_{i=1}^{n}i) = 1+\frac{1}{nm}(n^2-\frac{n(n+1)}{2}) = 1+\frac{n-1}{2m} = 1+\frac{\alpha}{2}-\frac{\alpha}{2n}.$$

Тому час, потрібний для виконання вдалого пошуку, включаючи час обчислення хеш-функції, буде $\Theta(2+\alpha/2-\alpha/(2n)) = \Theta(1+\alpha)$.

З цього випливає: для випадку $n = O(m)$ маємо $\alpha = O(1)$. Тому пошук елементу в хеш-таблиці у середньому буде вимагати $O(1)$ часу. А у найгіршому випадку вставка елементу до хеш-таблиці вимагає $O(1)$ часу, вилучення за використання двічі зв'язних списків також $O(1)$ часу.

Хеш-функції. Побудова некриптографічних хеш-функцій.

Перевірити виконання умови простого рівномірного хешування звичайно просто неможливо, так як невідомий розподіл ймовірностей,

відповідно з яким надходять ключі, що уводяться до таблиці. Крім того, ключі, що вставляють, можуть не бути незалежними.

При побудові якісних хеш-функцій дуже корисною є інформація про розподіл ключів. Вірним підходом при побудові є підбір хеш-функції так, щоб вона не корелювала із закономірностями, яким можуть підкорятися існуючі дані. Іноді до хеш-функцій можуть висувати більш суворі вимоги, ніж за простого рівномірного хешування.

Нехай сукупність ключів можна представити множиною цілих невід'ємних чисел N.

1. Побудова хеш-функції *методом ділення*: $h(k) = k \bmod m$ (відображення ключа k до одної з m комірок за допомогою вказаного обчислення залишку). Таке хешування доволі швидке. Але необхідно уникати деяких значень m (не повинно бути степенем 2, так як для $m = 2^p$ значення $h(k)$ буде просто p молодших бітів числа k, або 2^{p-1}). Часто m вибирають простим, досить далеким від числа, що є степенем 2.
2. Побудова хеш-функції *методом множення*: $h(k) = \lfloor m(kA \bmod 1) \rfloor$, $0 < A < 1$. В методі значення m може бути довільним, часто вибирають $m = 2^p$, де p – деяке натуральне, але деякі значення константи A дають кращі результати у порівнянні з іншими. Оптимальне значення константи A залежить від характеристик даних, що хешуються (але $A \approx (\sqrt{5}-1)/2 \approx 0.6180339887$ [2]).
3. Побудова хеш-функції з використанням *універсального хешування*. Будь-яка фіксована хеш-функція може стати вразливою, якщо підбираються дані, які будуть хешуватися до одної комірки та давати час пошуку $\Theta(n)$. Вказане спричинило ідею випадкового вибору хеш-функції (універсального хешування), який не залежить від того, з якими ключами потрібно працювати та *гарантує добру продуктивність у середньому* (за наведеною нижче теоремою). При використанні універсального хешування випадковим чином вибирають хеш-функцію з деякого обраного наперед класу функцій. Виконана рандомізація гарантує, що одні й ті ж вхідні дані не можуть постійно давати найгіршу поведінку алгоритму, та гарантує високу середню продуктивність для до-

вільних вхідних даних. Нехай H – скінчена множина хеш-функцій, які відображають дану сукупність ключів U у $\{0,1,...,m-1\}$. Множину H називають **універсальною**, якщо для кожної пари різних $k,l \in U$ кількість хеш-функцій $h \in H$, для яких $h(k) = h(l)$, не перевищує $|H|/m$ (тобто не перевищує ймовірності співпадіння двох випадковим чином вибраних хеш-значень з множини $\{0,1,...,m-1\}$, що дорівнює $1/m$).

Теорема. Нехай хеш-функція h, що випадковим чином вибрана з універсальної множини хеш-функцій, застосовується для хешування n ключів у таблицю T розміром m з використанням для вирішення колізій методу ланцюгів. Якщо ключ k відсутній у таблиці, то математичне очікування $E[n_{h(k)}]$ довжини списку, в який хешується ключ k, не перевищує коефіцієнту заповнення таблиці α (α = n/m). Якщо ключ k знаходиться у таблиці, то математичне очікування $E[n_{h(k)}]$ довжини списку, в якому знаходиться ключ k, не перевищує 1+α [1].

Доведення. Не уводячи припущення про розподіл ключів визначимо для кожної пари різних ключів $k,l \in U$ індикаторну випадкову величину $X_{kl} = I\{h(k) = h(l)\}$. За визначенням універсально множини H пара ключів викликає колізію із ймовірністю не більше $1/m$, тому $\Pr\{h(k) = h(l)\} \le 1/m$. А за основною властивістю індикаторної випадкової величини: $E[X_{kl}] \le 1/m$. Для кожного ключа k визначимо випадкову величину Y_k - кількість ключів, що відрізняються від k та хешуються до тієї ж комірки, що й ключ k. Тоді $Y_k = \sum_{l \in T, l \ne k} X_{kl}$ та

$$E[Y_k] = E\left[\sum_{\substack{l \in T \\ l \ne k}} X_{kl}\right] = \sum_{\substack{l \in T \\ l \ne k}} E[X_{kl}] \le \sum_{\substack{l \in T \\ l \ne k}} \frac{1}{m}.$$

Далі, якщо ключ k не знаходиться в таблиці T, то $n_{h(k)} = Y_k$ та $|\{l : l \in T \text{ й } l \ne k\}| = n$. Тому $E[n_{h(k)}] = E[Y_k] \le n/m = \alpha$.

Якщо ключ k знаходиться в таблиці T у списку $T[h(k)]$ та кількість ключів, що відрізняються від k та хешуються до списку, не включає

ключ k, то $n_{h(k)} = Y_k + 1$ та $\left|\{l : l \in T \text{ й } l \neq k\}\right| = n - 1$. Тому $E[n_{h(k)}] = E[Y_k] + 1 \leq (n-1)/m + 1 = 1 + \alpha - 1/m < 1 + \alpha$.

Наслідок (показує неможливість послідовності, що дасть найгірший час роботи – гарантується середній час роботи алгоритму за будь-яких вхідних даних). Використання універсального хешування та вирішення колізій методом ланцюгів у початково пустій таблиці з m комірками дає математичне очікування часу виконання довільної послідовності з n операцій Insert, Search, Delete, де міститься $O(m)$ Insert, розміру $\Theta(n)$.

Доведення. Оскільки кількість вставок є величиною розміру $O(m)$, то $n = O(m)$, а тому $\alpha = O(1)$. Час роботи операцій Insert, Delete є сталою величиною, за теоремою математичне очікування часу виконання кожної операції Search є $O(1)$. Тому математичне очікування часу, необхідного для виконання всієї послідовності з n операцій буде $O(n)$. Зважаючи на те, що кожна операція займає час $\Omega(1)$, одержимо границю з оцінкою $\Theta(n)$.

Побудова універсального класу хеш-функцій.

1. Виберемо просте число p, доволі велике, щоб всі можливі ключі знаходилися в межах від 0 до $p - 1$. Позначимо: $Z_p = \{0, 1, ..., p-1\}$, $Z_p^* = \{1, 2, ..., p-1\}$. Так як простір ключів є більшим, ніж кількість комірок у таблиці, то $p > m$. Визначимо хеш-функцію h_{ab} для довільних $a \in Z_p$ та $b \in Z_p^*$ як

$$h_{ab}(k) = ((ak + b) \bmod p) \bmod m.$$

Родина таких функцій утворює множину $H_{pm} = \{ h_{ab} : a \in Z_p \text{ та } b \in Z_p^* \}$. Кожна хеш-функція h_{ab} відображає множину Z_p в Z_m. Клас H_{pm} має $p(p-1)$ хеш-функцій.

Теорема. Клас H_{pm} хеш-функцій є універсальним [1].

Доведення. Розглянемо два різних ключа $k, l \in Z_p$, $k \neq l$. Нехай для даної хеш-функції h_{ab}

$$r = (ak + b) \bmod p, \qquad s = (al + b) (\bmod p).$$

Маємо, що $r \neq s$, $r - s \equiv a(k - l) (\bmod p)$, де p – просте число. Тому обчислення будь-якої хеш-функції h_{ab} з H_{pm} відображає різні вхідні

значення $k,l \in Z_p$ на різні значення r,s за модулем p (колізій за «модулем р» немає). Крім того, кожен з $p(p-1)$ можливих виборів пар (a, b), $a \neq 0$ дає різні пари (r,s) з $r \neq s$, оскільки за заданими r,s можемо знайти a, b:

$$a = ((r-s)(k-l)^{-1} (\mathrm{mod}\, p)) \bmod p, \quad b = (r - ak) \bmod p,$$

де $(k-l)^{-1} (\mathrm{mod}\, p)$ – мультиплікативне обернене за модулем p значення. Так як маємо $p(p-1)$ можливих пар (r,s) з $r \neq s$, то є взаємно однозначна відповідність між (a, b), $a \neq 0$ та (r,s) з $r \neq s$. Тому для довільної даної пари вхідних значень k,l за рівномірного випадкового вибору пари (a, b) з $Z_p^* \times Z_p$ одержана пара (r,s) може бути з рівною ймовірністю будь-якою з пар з різними за модулем p значеннями. Звідси одержимо, що ймовірність того, що різні ключі k,l приводять до колізії дорівнює ймовірності того, що $r \equiv s (\mathrm{mod} m)$ за випадкового вибору різних за модулем p значень r,s. Для заданого значення r маємо $p-1$ можливих значень s та число значень s для $r \neq s$ та $s \equiv r (\mathrm{mod} m)$ не перевищує

$$\lceil p/m \rceil - 1 \le ((p+m-1)/m) - 1 = (p-1)/m.$$

Ймовірність того, що s приводить до колізії з r при приведенні за модулем m, не перевищує

$$((p-1)/m)/(p-1) = 1/m.$$

Тому, для довільної пари різних значень $k,l \in Z_p$ $\Pr\{h_{ab}(k) = h_{ab}(l)\} \le 1/m$, тобто множина хеш-функцій H_{pm} є універсальною.

2. Нехай розмір хеш-таблиці визначений як просте число $p \approx m$. Універсальну множину будемо визначати векторами виду $x = (x_1, x_2, ..., x_r)$ для деякого цілого r, $0 \le x_i < p$ для кожного $i = 1,\ldots, r$. Наприклад, можна спочатку ідентифікувати множину U цілими числами з діапазону $[0, N-1]$ для деякого N, а потім скористатися суміжними блоками з $\lfloor \log p \rfloor$ бітів k для визначення відповідних координат x_i. Якщо $U \subseteq [0, N-1]$, то необхідна кількість координат буде $r \approx \log N^{\log m}$.

Нехай A – множина всіх векторів вигляду $a = (a_1, a_2, ..., a_r)$, де кожне $0 \le a_i \le p-1$ – ціле. Для кожного вектору $a \in A$ визначимо функцію $h_a(x) = (\sum_{i=1}^{r} a_i x_i) \bmod p$. Тоді родина хеш-функцій визначається як $H = \{ h_a : a \in A \}$.

При використанні хеш-функції h_a з класу визначеного класу H колізія $h_a(x) = h_a(l)$ визначає лінійне рівняння з обчисленням залишку від ділення на просте число p. Використаємо властивість, що для будь-яких простого числа p та цілого $z \ne 0 \bmod p$ та довільних цілих α, β, якщо $\alpha z = \beta z \bmod p$, то $\alpha = \beta \bmod p$ для доведення універсальності класу H.

Теорема. Клас лінійних хеш-функцій H є універсальним [1].

Доведення. Нехай $x = (x_1, x_2, ..., x_r)$ та $l = (l_1, l_2, ..., l_r)$ – два різні елемента з множини ключів U. Покажемо, що ймовірність $h_a(x) = h_a(l)$ для випадковим чином вибраного $a \in A$ не перевищує 1/m.

Так як $x \ne l$, то має існувати деякий індекс, наприклад, j, для якого $x_j \ne l_j$. Розглянемо наступний спосіб вибору випадкового вектора $a \in A$: спочатку вибираються всі координати a_i, для яких $i \ne j$, потім вибираються координати a_j. Покажемо, що незалежно від вибору всіх інших координат a_i ймовірність $h_a(x) = h_a(l)$, що обумовлена завершуючим вибором a_j дорівнює 1/m. З цього випливає, що ймовірність $h_a(x) = h_a(l)$ за випадкового вибору повного вектору a також має бути 1/m.

Нехай: B – подія $h_a(x) = h_a(l)$, C_b – подія одержання всіма координатами a_i, для яких $i \ne j$, послідовності значень b. Якщо $\Pr\{B / C_b\} = 1/m$ (покажемо нижче) для всіх b, то маємо:

$$\Pr\{B\} = \sum_b \Pr\{B / C_b\} \Pr\{C_b\} = 1/m \sum_b \Pr\{C_b\} = 1/m .$$

Припустимо, що значення всіх інших координат a_i були вибрані довільно, та розглянемо ймовірність такого вибору a_j, за якого $h_a(x) = h_a(l)$. Маємо $h_a(x) = h_a(l)$ тільки тоді, коли

$$a_j (l_j - x_j) = \sum_{i \ne j} a_i (l_i - x_i) \bmod p .$$

Так як варіанти вибору всіх a_i, для яких $i \neq j$, були фіксовані, то праву частину можна розглядати як фіксовану величину m. Визначимо $z = l_j - x_j$. Покажемо, що існує тільки одне значення $0 \le a_j < p$, за якого виконується $a_j z = m \bmod p$ (якщо це так, то ймовірність вибору цього значення a_j буде точно $1/m$). Припустимо, що таких значень два: a_j та a'_j. Тоді $a_j z = a'_j z \bmod p$ та маємо $a_j = a'_j \bmod p$. Але припустили, що $0 \le a_j < p$ та $0 \le a'_j < p$. Тому a_j та a'_j мають співпадати, тобто в цьому діапазоні є тільки одне a_j, за якого $a_j z = m \bmod p$.

Звідси випливає, що ймовірність вибору a_j, за якого $h_a(x) = h_a(l)$, дорівнює $1/m$ за будь-яких призначень інших координат a_i, для яких $i \neq j$, а тому ймовірність колізії між $x = (x_1, x_2, ..., x_r)$ та $l = (l_1, l_2, ..., l_r)$ буде $1/m$, тобто H – універсальний клас хеш-функцій.

Лекція 2. Метод відкритої адресації (закрите хешування). Ідеальне хешування

Метод відкритої адресації (закрите хешування). При використанні цього методу безпосередньо в хеш-таблиці зберігаються всі елементи, а якщо комірка пуста, то вона має містити значення NIL. Тому таблиця може стати повністю заповненою (коефіцієнт заповнення α не може перевищувати 1) і елемент вставити буде неможливо. При виконанні пошуку елементу перевіряються всі комірки таблиці, доки не буде знайдено шуканого елементу або впевнимося у відсутності елементу в хеш-таблиці. Перевагою методу є відмова від вказівників (економія пам'яті на них дозволяє збільшити розмір таблиці), але вже обчислюють послідовність комірок, що перевіряється.

Використовують методику *повторного хешування*. За нею для виконання вставки елементу послідовно перевіряють (досліджують) комірки таблиці, доки не знайдеться пуста комірка, в яку й розміщують ключ. Тобто, замість фіксованого порядку дослідження комірок 0, 1, …, m – 1 (що дає час $\Theta(n)$), послідовність досліджуваних комірок за-

лежить від ключа, що вставляють до хеш-таблиці. З цією метою використовується розширене визначення хеш-функції, де в якості другого аргументу включений номер дослідження (який починається з 0):

$$h: U \times \{0,1,..., m-1\} \to \{0,1,..., m-1\}.$$

В методі відкритої адресації потрібно, щоб для кожного ключа k послідовність досліджень $\langle h(k,0), h(k,1),..., h(k,m-1)\rangle$ являла собою перестановку множини $\langle 0,1,..., m-1\rangle$ та щоб могли бути переглянутими всі комірки таблиці.

Для випадку, коли елементи в хеш-таблиці T є ключами без супутньої інформації й ключ k тотожній елементу, що містить ключ k, процедура вставки ключа k може мати наступний вигляд [1]:

```
     HASH-ISERT(T, k)
1    i = 0
2    repeat
3         j = h(k, i)
4         if  T[j] == NIL
5              T[j] = k
6              return  j
7           else  i = i + 1
8    until   i == m
9    error "переповнення хеш-таблиці"
```

У процедурі пошуку ключа k досліджується та ж послідовність комірок, що й при його вставці. Але, якщо зустрічається пуста комірка, то пошук завершується невдачею (якщо припустимо, що вилучень не було) [1]:

```
     HASH-SEARCH(T, k)
1    i = 0
2    repeat
3         j = h(k, i)
4         if  T[j] == k
5              return  j
6           i = i + 1
```

7 **until** T[*j*] == *NIL* або *i* == *m*
8 **return** *NIL*

Процедура вилучення з хеш-таблиці більш складна. При вилученні ключа з комірки *i* хеш-таблиці її не можна просто помітити пустим значенням *NIL* (це негативно впливає на пошук елементу), таку комірку помічають спеціальним значенням DELETED. Крім того, потрібно дещо змінити наведену процедуру вставки, щоб в ній комірка із значенням DELETED розглядалася як пуста. Але час пошуку тоді перестає залежати від коефіцієнта заповнення α і тому при необхідності вилучень з таблиці користуються методом ланцюгів в якості методу вирішення колізій.

В методі відкритої адресації базовим є припущення про *рівномірне хешування*, а саме: для кожного ключа в якості послідовності досліджень рівноймовірні всі $m!$ перестановок множини $\langle 0,1,...,m-1\rangle$. Як узагальнення простого рівномірного хешування рівномірне хешування дає послідовність досліджень. Для реалізації рівномірного хешування використовують його апроксимації, але дійсно рівномірне хешування одержати важко.

Розглянемо три широко використовуваних методи обчислення послідовності досліджень, які гарантують: $\langle h(k,0),h(k,1),...,h(k,m-1)\rangle$ є перестановкою для кожного ключа *k*. Недоліком цих методів є те, що вони не задовольняють припущенню про рівномірне хешування, так як жоден не може згенерувати більше m^2 різних послідовностей досліджень.

1. *Лінійне дослідження*. Нехай задана в якості допоміжної хеш-функції функція $h':U\to\{0,1,...,m-1\}$. Для обчислення послідовності досліджень в методі використовується хеш-функція $h(k,i)=(h'(k)+i)\bmod m$ для $i = 0,1,\ldots, m-1$. Для деякого ключа *k* першою досліджуваною коміркою є $T[h'(k)]$, наступними – $T[h'(k)+1]$, …, $T[m-1]$, $T[0]$, …, останньою – $T[h'(k)-1]$. Так як перша досліджувана комірка єдиним чином визначає повністю всю послідовність досліджень, то усього маємо m різних послідовностей. Також недоліком при використанні лінійного дослідження

є виникнення первинної кластеризації, що пов'язана з утворенням довгих послідовностей зайнятих комірок, що збільшує середній час пошуку. Кластери виникають внаслідок того, що ймовірність заповнення пустої комірки, якій передує i заповнених комірок, дорівнює $(i + 1)/m$. Тому довгі серії заповнених комірок мають тенденцію до ще більшого зростання, що збільшує середній час пошуку [1].

2. *Квадратичне дослідження*. Для обчислення послідовності досліджень використовують хеш-функцію $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$ для $i = 0,1,\ldots, m-1$ (h' – допоміжна хеш-функція, c_1 та c_2 – додані допоміжні сталі). Для деякого ключа k першою досліджуваною коміркою є $T[h'(k)]$, наступною – $T[(h'(k) + c_1 + c_2) \bmod m]$ і т.д. Але щоб дослідження охоплювало всі комірки, необхідно спеціальним чином обирати сталі c_1, c_2, m. Крім того, якщо два ключі матимуть однакову початкову позицію дослідження, то вони будуть мати й повністю однакові послідовності, а це приводить до вторинної кластеризації. Тут теж, оскільки перша досліджувана комірка єдиним чином визначає повністю всю послідовність досліджень, то усього маємо m різних послідовностей [1].
3. *Подвійне хешування*. Для обчислення послідовності досліджень використовується хеш-функція $h(k,i) = (h_1(k) + i h_2(k)) \bmod m$ для $i = 0,1,\ldots, m-1$ ($h_1(k), h_2(k)$ – допоміжні хеш-функції). Для деякого ключа k першою досліджуваною коміркою є $T[h_1(k)]$, наступною – $T[(h_1(k) + h_2(k)) \bmod m]$ і т.д. В даному випадку послідовність дослідження залежить від ключа k за двома параметрами – вибору початкової комірки та відстані між сусідніми досліджуваними комірками. Щоб послідовність дослідження охоплювала всі комірки таблиці, необхідно щоб значення $h_2(k)$ було взаємно простим з розміром таблиці m. Забезпечити це може, наприклад, вибір значення m як числа степеню 2 та такої хеш-функції $h_2(k)$, що повертала б тільки непарні значення. Або можливо вибирати значення m як просте число та таку хеш-функцію $h_2(k)$, що повертала б тільки натуральні числа, що менші за m (наприклад, $h_1(k) = k \bmod m$, $h_2(k) = 1 + k \bmod m'$, де m' є трохи меншим за m).

Так як кожна пара хеш-функцій дає відмінну від інших послідовність досліджень, то подвійне хешування має $\Theta(m^2)$ послідовностей досліджень [1].

Оцінимо кількість досліджень, що необхідно виконати для пошуку елементу при використанні відкритої адресації. В аналізі використаємо коефіцієнт заповнення таблиці α, що в даному випадку не може перевищувати 1, та припущення рівномірного хешування. За припущення рівномірного хешування для кожного ключа *k* послідовність досліджень $\langle h(k,0), h(k,1), ..., h(k,m-1)\rangle$ з рівною ймовірністю є одною з можливих перестановок множини $\langle 0,1,...,m-1\rangle$. А так як з конкретним ключем *k* зв'язана єдина фіксована послідовність досліджень, то й всі послідовності досліджень будуть рівноймовірними.

Теорема. Математичне очікування кількості досліджень *за невдалого пошуку* в хеш-таблиці з відкритою адресацією та коефіцієнтом заповнення $\alpha = n/m < 1$ у припущенні рівномірного хешування не перевищує $1/(1-\alpha)$ [1].

Доведення. За невдалого пошуку кожна послідовність досліджень завершується пустою коміркою. Визначимо: випадкова додатна цілочисельна величина X – кількість досліджень, виконаних за невдалого пошуку; події A_i – виконане *i*-те дослідження та воно прийшлося на зайняту комірку ($i = 1, 2, \ldots$). Подія $\{X \ge i\}$ буде перетином подій A_j, $j = 1, 2, \ldots, i-1$ ($A_1 \cap A_2 \cap ... \cap A_{i-1}$). Обмежимо ймовірність $\Pr\{X \ge i\}$ шляхом обмеження $\Pr\{A_1 \cap A_2 \cap ... \cap A_{i-1}\}$. Маємо,

$$\Pr\{A_1 \cap A_2 \cap ... \cap A_{i-1}\} = \\ = \Pr\{A_1\}\Pr\{A_2 / A_1\}\Pr\{A_3 / A_1 \cap A_2\}...\Pr\{A_{i-1} / A_1 \cap A_2 \cap ... \cap A_{i-2}\}.$$

Так як усього n елементів, що розташовують у m комірках, то $\Pr\{A_1\} = n/m$. Вважається, що комірки вибираються випадковим чином, тому ймовірність колізії на початку, як вказано, дорівнює n/m.

Якщо комірка виявилася зайнятою, то на етапі повторного хешування вже працюємо з $m-1$ комірками, де розміщено вже $n-1$ елементів. Тоді ймовірність одразу двох колізій буде $(n(n-1))/(m(m-1))$.

Проводимо далі повторне хешування. Ймовірність, що буде виконане *j*-те дослідження, $j = 2, \ldots, i-1$, й що воно буде проведене над заповненою коміркою (а значить й перші $j-1$ досліджень проведені

над заповненими комірками) дорівнюватиме $(n - (j - 1))/ (m - (j - 1)) = (n - j + 1)/ (m - j + 1)$.

Якщо виявилося i колізій підряд, то ймовірність такого становить: $(n(n - 1)\ldots(n - (i - 1)))/(m(m - 1)\ldots(m - (i - 1)))$.

Так як з $n < m$ для всіх $0 \le j < m$ справедливе $(n - j)/(m - j) \le n/m$, то для всіх $1 \le i < m$:

$$\Pr\{X \ge i\} = \frac{n}{m}\frac{n-1}{m-1}\frac{n-2}{m-2}\ldots\frac{n-i+2}{m-i+2} \le \left(\frac{n}{m}\right)^{i-1} = \alpha^{i-1}.$$

Математичне очікування кількості досліджень за невдалого пошуку:

$$E[X] = \sum_{i=1}^{\infty} \Pr\{X \ge i\} \le \sum_{i=1}^{\infty} \alpha^{i-1} = \sum_{i=0}^{\infty} \alpha^{i} = \frac{1}{1-\alpha}.$$

Якщо α є сталою, то теорема вказує, що невдалий пошук виконується за час $O(1)$, що є оцінкою продуктивності відповідної процедури. Так, якщо, наприклад, α = ½, то середня кількість досліджень за невдалого пошуку не перевищуватиме 1/(1 – ½) = 2.

Наслідок. *Вставка елементу* в хеш-таблицю з відкритою адресацією та коефіцієнтом заповнення α у припущенні рівномірного хешування *вимагає у середньому* не більше 1/(1 – α) досліджень.

Доведення. Елемент може бути вставлений до хеш-таблиці тільки тоді, коли є вільне місце в таблиці, тобто α < 1. Вставка ключа вимагає проведення невдалого пошуку, за яким відбувається розміщення ключа у знайдену пусту комірку. Тому математичне очікування кількості досліджень не перевищуватиме 1/(1 – α) досліджень.

Теорема. Математичне очікування кількості досліджень *за вдалого пошуку* в хеш-таблиці з відкритою адресацією та коефіцієнтом заповнення α < 1 за припущення рівномірного хешування та рівноймовірного пошуку будь-якого з ключів не перевищує $\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$ [1].

Доведення. Пошук ключа k виконується тією ж послідовністю досліджень, що й вставка. За наслідком з попередньої теореми, якщо ключ k був $(i + 1)$-м ключем, що вставлений до хеш-таблиці, то математичне очікування кількості проб при пошуку ключа k не перевищує $1/(1 - i/m) = m/(m - i)$. Усереднення за всіма n ключами в хеш-таблиці дасть середню кількість досліджень за вдалого пошуку:

$$\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i}=\frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i}=\frac{1}{\alpha}\sum_{k=m-n+1}^{m}\frac{1}{k}\le\frac{1}{\alpha}\int_{m-n}^{m}\frac{1}{x}dx=\frac{1}{\alpha}\ln\frac{m}{m-n}=\frac{1}{\alpha}\ln\frac{1}{1-\alpha}.$$

При застосуванні методу відкритої адресації (ще кажуть – схеми закритого хешування) швидкість виконання операцій вставки та ін. залежить не тільки від рівномірності розподілу елементів по комірках обраною хеш-функцією, а й від обраної методики повторного хешування (заповнення) для вирішення колізій, що пов'язані із спробами вставки елементів до заповнених комірок. Як тільки декілька послідовних сегментів будуть заповнені (буде утворена група), то будь-який новий елемент при вставці буде розміщений у кінець групи, збільшуючи її довжину. Тепер для пошуку пустої комірки (наприклад, як за лінійного дослідження) потрібно буде переглянути більше комірок, ніж у випадку випадкового розподілу заповнених комірок. Це спричинить зростання часу виконання операцій (наприклад, вставки елемента). У зв'язку з цим цікавим є аналіз кількості перевірок на заповненість комірок при вставці елементу у припущенні, що в хеш-таблиці з m комірками вже є N елементів та всі комбінації розташування N елементів в комірках таблиці рівноймовірні. За наведеними теоремами середнє число спроб буде приблизно рівним (не перевищуватиме) $m/(m - \mathrm{N})$. Якщо N є ½m, то у середньому потрібно 2 спроби для вставки нового елементу. Для повного заповнення таблиці середнє число спроб на 1 комірку буде

$$\frac{1}{m}\sum_{i=0}^{m-1}\frac{m+1}{m+1-i}\le\frac{1}{m}\int_{0}^{m-1}\frac{m}{m-x}dx=\ln m$$

або всього $m\ln m$ спроб. Але для заповнення таблиці на 90 % потрібно буде всього m((10/9)1n10), або приблизно 2.56 спроб на 1 комірку.

При пошуку елементу, якого немає в таблиці, потрібна у середньому така ж кількість спроб, як і при вставці нового елементу. Але пошук елементу, який знаходиться в таблиці, вимагає у середньому стільки спроб, скільки необхідно для вставки всіх елементів, що зроблені до поточного часу. Вилучення вимагає у середньому стільки ж спроб, скільки й перевірка елементу на знаходження його в таблиці. Крім того, вилучення не прискорює процесу вставки нового елементу або перевірки знаходження елементу у таблиці. Середні кількості спроб, що необхідно зробити для виконання операцій, залежать від коефіцієнту заповнення хеш-таблиці (рис. 2.1).

З рис. 2.1 бачимо, що середній час виконання операцій зростає із зростанням α. Щоб зберегти постійний час виконання операцій можна при досягненні деякого значення коефіцієнту заповнення (наприклад, α = 0,9 або менше) проводити реструктуризацію хеш-таблиць – створю

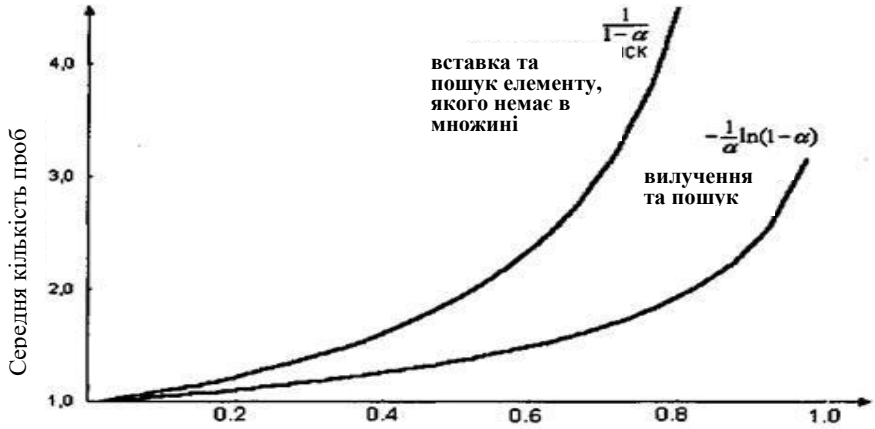


Рис. 2.1. Середні кількості проб, що необхідні для виконання основних операцій

нову хеш-таблицю, наприклад, у 2 рази більшу (попередньо провівши амортизаційний аналіз доцільності саме такого збільшення). Перезапис поточних елементів у нову хеш-таблицю у середньому займе менше часу, ніж раніше виконана їх вставка до старої хеш-таблиці та буде компенсований швидким виконанням основних операцій.

Ідеальне хешування. Використовують з метою забезпечення гарної продуктивності у найгіршому випадку для статичної множини ключів (коли всі ключі збережені в таблиці та їх множина ніколи не змінюється: множина імен файлів на диску та ін.).

Ідеальним хешуванням називають метод хешування, що виконує пошук за $O(1)$ звернень до пам'яті у найгіршому випадку.

Для створення схеми ідеального хешування використаємо двохрівневу схему хешування з універсальним хешуванням на кожному рівні. Перший рівень такий же, як у випадку хешування з ланцюгами. На ньому n ключів хешуються до m комірок з використанням хеш-функції h, що вибрана з родини універсальних хеш-функцій. Але замість того, щоб

створювати список ключів, що розміщені до комірки, наприклад, з номером j, створюється вторинна хеш-таблиця S_j (меншого розміру) із зв'язаною з ней хеш-функцією h_j. Шляхом акуратного вибору хеш-функції h_j можемо гарантувати відсутність колізій на другому рівні.
Щоб гарантувати відсутність колізій на другому рівні, потрібно, щоб розмір m_j хеш-таблиці S_j дорівнював квадрату числа n_j ключів, хешованих до комірки j. Вказане дозволяє за підходящого вибору хеш-функції першого рівня обмежити очікувану кількість необхідної для хеш-таблиці пам'яті значенням $O(n)$.
Хеш-функцію першого рівня виберемо з класу H_{pm}, де p – просте число, що перевищує значення будь-якого з ключів. Ключи, що хешовані до комірки j, другий раз хешуються до вторинної хеш-таблиці S_j розміром m_j з використанням хеш-функції h_j, що вибирається з класу H_{pmj} (якщо до комірки j попаде лише один елемент, то хеш-функція h_j не вибирається, для цього просто беремо $a = b = 0$).
Теорема. Нехай n ключів зберігається в хеш-таблиці розміром $m = n^2$ з використанням хеш-функції h, що випадково вибрана з універсального класу хеш-функцій. Тоді ймовірність виникнення колізій буде менше ½ [1].
Доведення. Маємо всього C_n^2 пар ключів, що можуть викликати колізію. Якщо хеш-функція вибрана випадково з універсального класу хеш-функцій H, то для кожної пари ймовірність виникнення колізії дорівнює $1/m$. Нехай X – додатна цілочисельна випадкова величина, що підраховує кількість колізій. Якщо $m = n^2$, то математичне очікування числа колізій:

$$E[X] = C_n^2 \frac{1}{n^2} = \frac{n(n-1)}{2}\frac{1}{n^2} < \frac{1}{2}.$$

Використаємо нерівність Маркова для $t = 1$: $\Pr\{X \geq t\} \leq E[X]/t$, отже ймовірність появи колізій буде менше ½.

Одержане значить, що для $m = n^2$ довільно вибрана з множини H хеш-функція з більшою ймовірністю не буде викликати колізії, ніж буде. Для деякої заданої статичної множини K, що містить n ключів, знайти хеш-функцію h, що не приводить до колізій, можна за декілька випадкових спроб.

Але, якщо значення n є великим, то й таблиця розміром $m = n^2$ буде надто великою. Внаслідок цього і використовують двохрівневу схему хешування, де результати наведеної теореми застосовуються до окремих комірок. На першому рівні зовнішня хеш-функція h, що вибрана з родини універсальних хеш-функцій, використовується для n ключів до $m = n$ комірок. Далі, якщо в комірку j хешовано n_j ключів, то для забезпечення відсутності колізій та пошуку за константний час, використовується вторинна хеш-таблиця S_j розміром $m_j = n^2_j$. Оскільки розмір m_j вторинної хеш-таблиці S_j зростає із зростанням n_j квадратично, то виникає питання про загальний розмір пам'яті. Якщо хеш-таблиця першого рівня має розмір $m = n$, то виявляється, що пам'яті потрібно $O(n)$.

Теорема. Нехай зберігається n ключів у хеш-таблиці розміром $m = n$ з використанням хеш-функція h, що випадково вибрана з класу універсальних хеш-функцій. Тоді $E\left[\sum_{j=0}^{m-1} n_j^2\right] < 2n$, де n_j – кількість ключів, що хешована до комірки j [1].

Доведення. Використовуючи вираз $a^2 = a + 2\dfrac{a(a-1)}{2} = a + 2C_a^2$, справедливий для довільного цілого $a > 0$, маємо

$$E\left[\sum_{j=0}^{m-1} n_j^2\right] = E\left[\sum_{j=0}^{m-1} (n_j + 2C_{n_j}^2)\right] = E\left[\sum_{j=0}^{m-1} n_j\right] + 2E\left[\sum_{j=0}^{m-1} (C_{n_j}^2)\right] =$$

$$= E[n] + 2E\left[\sum_{j=0}^{m-1} (C_{n_j}^2)\right] = n + 2E\left[\sum_{j=0}^{m-1} (C_{n_j}^2)\right].$$

Щоб обчислити другий доданок, звернемо увагу, що вираз, який сумується в ньому, задає загальну кількість колізій. За властивістю універсального хешування математичне очікування цієї суми не перевищує $C_n^2 \dfrac{1}{m} = \dfrac{n(n-1)}{2m} = \dfrac{(n-1)}{2}$. Тому $E\left[\sum_{j=0}^{m-1} (n_j^2)\right] \le n + 2\dfrac{n-1}{2} = 2n - 1 < 2n$.

Наслідок. Нехай зберігається n ключів у хеш-таблиці розміром $m = n$ з використанням хеш-функція h, що випадково вибрана з класу універсальних хеш-функцій, та встановлюється розмір кожної вторинної хеш-таблиці рівний $m_j = n^2_j$, $j = 0, 1, \ldots, m - 1$, тоді *математичне очікування кількості необхідної пам'яті* для *вторинних хеш-таблиць* у схемі ідеального хешування *не перевищує величини* $2n$.

Доведення. Так як $m_j = n^2_j$, $j = 0, 1, \ldots, m - 1$, то за теоремою $E\left[\sum_{j=0}^{m-1} m_j\right] = E\left[\sum_{j=0}^{m-1} n_j^2\right] < 2n$.

Наслідок. Нехай зберігається n ключів у хеш-таблиці розміром $m = n$ з використанням хеш-функція h, що випадково вибрана з класу універсальних хеш-функцій, та встановлюється розмір кожної вторинної хеш-таблиці рівний $m_j = n^2_j, j = 0, 1, \ldots, m - 1$, тоді *ймовірність* того, що *загальна кількість необхідної пам'яті для вторинних хеш-таблиць не менше 4n* буде меншою, ніж ½.

Доведення. Застосуємо нерівність Маркова для $X = \sum_{j=0}^{m-1} m_j$ та $t = 4n$:

$$\Pr\left\{\sum_{j=0}^{m-1} m_j \geq 4n\right\} \leq E\left[\sum_{j=0}^{m-1} m_j\right] \Big/ 4n < \frac{2n}{4n} = 1/2.$$

З наслідку випливає, що, якщо перевірити декілька випадковим чином обраних з універсального класу хеш-функцій, доволі швидко знайдемо хеш-функцію, що забезпечить помірні вимоги до кількості пам'яті.

Лекція 3. Застосування некриптографічних хеш-функцій. Поліноміальні хеш-функції

Існує два підходи у виборі поліноміальної хеш-функції: напрям зростання степенів основ зліва-направо та напрям справо-наліво.
Нехай заданий деякий рядок $S_{0..n-1}$ з n символів.

Поліноміальним хешем (поліноміальною хеш-функцією, функцією поліноміального хешу) у варіанті зростання степенів у поліномі *зліва-направо* для послідовності символів $\{a_0, a_1, ..., a_{n-1}\}$ називають хеш-функцію

$$h(a, p, m) = (a_0 + a_1 p + a_2 p^2 + ... + a_{n-1} p^{n-1}) \bmod m,$$

де p – точка (основа), що є деяким натуральним числом, взаємно простим з m, m – модуль хешування (наприклад, 2^{64}), що задовольняють нерівності $\max(a_i) < p < m$, a_i – код i-го символу рядку S.

Тобто хеш-функція співставляє послідовності $\{a_0, a_1, ..., a_{n-1}\}$ число довжини n у системі числення p та задає залишок від його ділення на число m, або значення багаточлена $(n-1)$-ї степені з коефіцієнтами a_i в точці p за модулем m (якщо значення $h(a, p, m)$, не взяте за модулем, вміщується у цілочисельний тип даних (наприклад, 64-бітний тип), то можемо кожній послідовності співставити це число, а тоді порівняння типу більше, менше, дорівнює можна виконувати за $O(1)$).)

Для такого порівняння достатньо порахувати поліноміальну хеш-функцію на кожному префіксі початкової послідовності $\{a_0, a_1, ..., a_{n-1}\}$. У однакових рядків значення хеш-функції рівні. Якщо значення хеш-функцій будуть рівні, то можна зробити висновок про рівність підпослідовностей однакової довжини з дуже великою ймовірністю (можливе хибне спрацювання).

Розглянемо як порівняти довільні неперервні підвідрізки послідовності за $O(1)$ (звичайно для порівняння двох рядків довжини n у найгіршому випадку необхідно перевірити усі n символів, а це $O(n)$). Так як значення хеш-функції – це число, то для порівняння підвідрізків потрібно $O(1)$ часу. Але щоб порахувати хеш-функцію одного рядку в лоб потрібно $O(n)$ часу. Тому за $O(n)$ шукають хеші одразу усіх підрядків. Якими підходами це можна зробити?

1. Визначимо поліноміальну хеш-функцію на префіксі наступним чином:

$$h_{pref}(k) = h_{pref}(k, a, p, m) = (a_0 + a_1 p + a_2 p^2 + ... + a_{k-1} p^{k-1}) \bmod m.$$

Маємо:

$$h_{pref}(0) = 0,\ h_{pref}(1) = a_0,\ h_{pref}(2) = (a_0 + a_1 p) \bmod m,$$

…

$$h_{pref}(n) = (a_0 + a_1 p + a_2 p^2 + ... + a_{n-1} p^{n-1}) \bmod m.$$

Тоді, для знаходження поліноміальної хеш-функції на кожному префіксі за час $O(n)$, можна використати наступні рекурентні співвідношення (за вказаної вище умови):

$$p^k = p^{k-1} p,\ \ h_{pref}(k) = h_{pref}(k-1) + a_k p^{k-1}.$$

Крім того,

$$h_{pref}(n) = h_{pref}(k-1) + p^k h_{pref}(k..n),$$

де $h_{pref}(k..n) = a_k p^k + a_{k+1} p^{k+1} + ... + a_{n-1} p^{n-1}$.

Якщо потрібно порівняти на рівність два підрядки однакової довжини *len*, які починаються в позиціях *i* та *j*, то потрібно розглянути різниці $h_{pref}(i+len) - h_{pref}(i)$ та $h_{pref}(j+len) - h_{pref}(j)$.

Маємо:

$$h_{pref}(i+len) - h_{pref}(i) = a_i p^i + a_{i+1} p^{i+1} + ... + a_{i+len-1} p^{i+len-1}, \quad (3.1)$$

$$h_{pref}(j+len) - h_{pref}(j) = a_j p^j + a_{j+1} p^{j+1} + ... + a_{j+len-1} p^{j+len-1}. \quad (3.2)$$

Приведемо ці рівняння до одного степеня за основою *p*, для цього домножимо справа й зліва на величини p^j та p^i, відповідно:

$$\left(h_{pref}(i+len) - h_{pref}(i)\right)p^j = \left(a_i + a_{i+1} p + ... + a_{i+len-1} p^{len-1}\right)p^{i+j},$$

$$\left(h_{pref}(j+len) - h_{pref}(j)\right)p^i = \left(a_j + a_{j+1} p + ... + a_{j+len-1} p^{len-1}\right)p^{j+i}.$$

В одержаних рівняннях в правій частині в дужках маємо поліноміальні хеш-функції підвідрізків $a_i, a_{i+1},...,a_{i+len-1}$ та $a_j, a_{j+1},...,a_{j+len-1}$. Щоб визначити чи співпадають ці підвідрізки, маємо перевірити виконання такої рівності:

$$\left(h_{pref}(i+len) - h_{pref}(i)\right)p^j = \left(h_{pref}(j+len) - h_{pref}(j)\right)p^i.$$

Одне таке порівняння можна виконати за час $O(1)$, якщо наперед визначати степені *p* за модулем *m* (використовуючи рекурентний вираз $p^k = p^{k-1} p$). З урахуванням модуля *m* попередня рівність прийме вигляд:

$$p^j \left(h_{pref}(i+len) - h_{pref}(i)\right) \bmod m = p^i \left(h_{pref}(j+len) - h_{pref}(j)\right) \bmod m. \quad (3.3)$$

Але рівність (3.3) показує, що порівняння рядків залежить від параметрів порівнюваних з ними рядків. Для вирішення цієї проблеми рівняння (3.1) множать на p^{-i}, а (3.2) – на p^{-j}. Тоді маємо:

$$\left(h_{pref}(i+len) - h_{pref}(i)\right)p^{-i} = a_i + a_{i+1} p + ... + a_{i+len-1} p^{len-1},$$

$$\left(h_{pref}(j+len) - h_{pref}(j)\right)p^{-j} = a_j + a_{j+1} p + ... + a_{j+len-1} p^{len-1}.$$

В правих частинах отримаємо хеш-функції для підвідрізків, що дає підставу для наступного рівняння для перевірки рівності цих підвідрізків:

$$p^{-i}\left(h_{pref}(i+len) - h_{pref}(i)\right) = p^{-j}\left(h_{pref}(j+len) - h_{pref}(j)\right) \quad (3.4)$$

Для реалізації такого підходу необхідно знайти обернений елемент для *p* за модулем *m*, а так як найбільшим спільним дільником цих чисел є

1, то такий елемент існує. Якщо наперед порахувати степені оберненого елемента за модулем, то порівняння можемо виконати за $O(1)$.

2. Нехай *Mrow* – максимальна довжина порівнюваних підрядків. Домножимо рівняння (3.1) на p в степені $Mrow - i - len + 1$, а рівняння (3.2) – на p в степені $Mrow - j - len + 1$:

$$p^{Mrow-i-len+1}\left(h_{pref}(i+len)-h_{pref}(i)\right)=p^{Mrow-len+1}\left(a_i+a_{i+1}p^1+...+a_{i+len-1}p^{len-1}\right), \quad (3.5)$$

$$p^{Mrow-j-len+1}\left(h_{pref}(j+len)-h_{pref}(j)\right)=p^{Mrow-len+1}\left(a_j+a_{j+1}p^1+...+a_{j+len-1}p^{len-1}\right). \quad (3.6)$$

В рівняннях (3.5), (3.6) справа містяться поліноміальні хеші для заданих підрядків, тому рівність цих підрядків можемо перевірити з наступного рівняння:

$$p^{Mrow-i-len+1}\left(h_{pref}(i+len)-h_{pref}(i)\right)\bmod m = p^{Mrow-j-len+1}\left(h_{pref}(j+len)-h_{pref}(j)\right)\bmod m. \quad (3.7)$$

Наведене рівняння дозволяє порівнювати підрядок довжини *len* з іншими підрядками такої ж довжини, причому вони можуть бути підрядками іншого рядку, так як вирази справа та зліва в рівнянні залежать тільки від власних параметрів розглянутого підрядку (i, *len*) та сталої *Mrow*.

Отже, для обчислення хешу будь-якого підрядка $h_{pref}(n)$ необхідно знати хеш-функції префіксів підрядків $h_{pref}(0..0)$, $h_{pref}(0..1)$, $h_{pref}(0..2)$, …, $h_{pref}(0..n-1)$:

$$h_{pref}(0..0) = a_0,$$

…,

$$h_{pref}(0..k) = (h_{pref}(0..k-1)\,p + a_k) \bmod m,\ p(k) = p^k \bmod m, \ldots .$$

Так як хеш кожного наступного префіксу виражається через хеш попереднього, то вважатимемо це лінійним проходом за рядком. Усе разом зможемо обрахувати за $O(n)$ часу. Степені p рахуються наперед і зберігаються у масиві.

У загальному алгоритм може бути наступним:

1. Зберігаємо обчислені p^i у масиві.
2. Обчислюємо хеш-функції префіксів першого рядку.
3. Обчислюємо хеш-функції префіксів другого рядку.
4. Порівнюємо 2 рядки за $O(1)$.

Але, так як рядки можуть бути дуже довгими, то при обчисленні хеш-функції можемо отримати переповнення. Тому хеші рахують, наприклад, за mod 2^{64}, $2^{31} - 1$, $p > \max\{a_i\}$ взаємно просте з m, непарне (якщо маємо справу лише з прописними символами латинської абетки, наприклад, $p = 31$). Так як всі операції виконуються за mod, то для позбавлення від ділення вирази часто приводять до загального знаменника і замість ділення використовують множення (наприклад, вираз (3.7)). Але можна й використовувати вираз типу (3.4). Одне таке порівняння виконуємо за $O(1)$.

Оцінимо ймовірність колізій при використанні поліноміальної хеш-функції.

Нехай потрібно обчислювати хеш за модулем m та проводимо порівняння n рядків. Використовуючи парадокс днів народження одержимо ймовірність того, що відбудеться колізія: $p \approx 1 - \exp(\frac{n^2}{2m})$.

Наведене показує, що величину m потрібно брати значно більше за n^2.

Тоді $p \approx \frac{n^2}{2m}$, але ймовірність не враховує зламу.

Подвійний поліноміальний хеш. Нехай маємо надто великі значення для модуля m. Але, якщо візьмемо два взаємно простих модуля m_1 та m_2, то кільце залишків за модулем $m = m_1 \cdot m_2$ еквівалентне добутку кілець за модулями m_1 та m_2, тобто між ними існує взаємно однозначна відповідність, основана на ідемпотентах кільця вичетів за модулем m. Тобто, якщо обчислювати h за модулем m_1 та за модулем m_2, а потім порівнювати два підвідрізки.

Для *порівняння* двох підрядків довжиною n, необхідно спочатку перевірити на співпадіння довжини відрізків, обчислити p^i, $i = 0, \ldots, n$-1, у масиві, перерахувати хеш-функції для всіх l, $l = 0, \ldots,$ N-n, як

$$h_{pref}(l+1..l+n-1) = (h_{pref}(l..l+n-1) - a_l p^n) \bmod m,$$

$$h_{pref}(l+1..l+n) = (p h_{pref}(l+1..l+n-1) + a_{l+n}) \bmod m.$$

Звідси

$$h_{pref}(l+1..l+n) = ((h_{pref}(l..l+n-1) - a_l p^{n-1}) p + a_{l+n}) \bmod m. \quad (3.8)$$

Вираз (3.8) є основою для алгоритму порівняння підрядків, але внаслідок наявності можливих хибних спрацювань, маємо ще проводити перевірку результатів (асимптотична оцінка $O(n)$).

Для *пошуку підрядку* довжиною n у рядку довжиною N знаходимо хеш підрядку та хеші всіх префіксів рядку, а потім рухаємося по рядку вікном довжини n порівнюючи хеші $h_{pref}(i..i+n-1)$.

Для *знаходження* z-функції (z-функція рядку довжиною n – це масив z, i-й елемент якого дорівнює найбільшому числу символів, починаючи з позиції i у рядку, що співпадають з першими символами рядку (асимптотична оцінка $O(n\log(n))$ для $i = 0, 1, \ldots, n\text{-}1$ шукаємо $z[i]$ бінарним пошуком.

Для *пошуку лексиграфічно мінімального циклічного зсуву рядку* (асимптотична оцінка $O(n\log(n))$ спочатку береться сам рядок за кращий (лексиграфічно мінімальний), потім для кожного циклічного зсуву за допомогою бінарного пошуку береться довжина максимального загального префіксу цього зсуву та поточної кращої відповіді. Далі порівнюються наступні за цим префіксом символи та, якщо потрібно, оновлюється відповідь. Якщо довжина цього префіксу дорівнює n, то поточний циклічний зсув співпадає з мінімальним та відповідь не потрібно поновлювати.

Для *пошуку усіх паліндромів* у рядку (асимптотична оцінка $O(n\log(n))$, тобто для пошуку підрядків $S_{L..R}$, в яких $a_L=a_R$, $a_{L+1}=a_{R-1}$, …, поруч із масивом, який містить хеші $h_{pref}(0..0)$, $h_{pref}(0..1)$, $h_{pref}(0..2)$, …, $h_{pref}(0..n-1)$ одержимо подібний масив з хешами для оберненого рядку. Розглянемо позицію i у рядку. Нехай існує паліндром непарної довжини d з центром у позиції i (парна – позиція між $i-1$, i). Якщо обрізати його по краях на 1 символ, він залишиться паліндромом. Діємо так, доки довжина не стане 0. В результаті достатньо для кожної позиції зберігати два значення: скільки існує паліндромів непарної довжини з центром у позиції i та скільки існує їх парної довжини з центром між між $i-1$, i.

Підхід з використанням напряму зростання степенів основ p, тобто *справо-наліво* (використовує алгоритм Рабіна-Карпа [1]) встановлює наступне поняття поліноміальної хеш-функції.

Поліноміальним хешем для рядку $S_{0..n}$ з $n + 1$ символів у варіанті зростання степенів у поліномі справо-наліво для послідовності символів $\{a_0, a_1, ..., a_n\}$, називають хеш-функцію

$$h(a, p, m) = (a_0 p^n + a_1 p^n + a_2 p^{n-2} + ... + a_n) \bmod m,$$

m – модуль хешування, що є фіксованим простим доволі великим числом (наприклад, $2^{31} - 1$, $2^{61} - 1$, $2^{32} - 5$, $2^{64} - 59$), основу p вибирають випадковим чином перед роботою алгоритму з діапазону від 0 до $m - 1$. Наведений у визначенні вираз запишемо у наступному вигляді:

$$h(a, p, m) = (...((a_0 p + a_1) p + a_2) p + ... + a_{n-1}) p + a_n) \bmod m.$$

Вказане дає можливість одержати наступне:

$$h_{pref}(i+1..i+n) = ((h_{pref}(i..i+n-1) - a_i p^{n-1}) p + a_{i+n}) \bmod m. \quad (3.9)$$

Вираз (3.9) і є основою для алгоритму порівняння підрядків.

Узагальнення поліноміальних хеш-функцій на двовимірні простори. Замість одної основи (числа p) та масиву степенів числа p в даному випадку використовують вже дві основи (p, q) та, відповідно, два масиви значень степенів. Хеш-функцією матриці елементів $A_{0..n\text{-}1,0..m\text{-}1}$ буде сума добутків:

$$h(A) = \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} a_{ij} p^i q^j .$$

Тобто, маємо такі хеш-функції префіксів підрядків:

$h_{pref}(a_{0..0,0..0}) = a_{00}$,

для всіх $i = 1,\ldots, n\text{-}1$ $h_{pref}(a_{0..i,0..0}) = h_{pref}(a_{0..i-1,0..0}) + p^i a_{i0}$,

для всіх $j = 1,\ldots, m\text{-}1$ $h_{pref}(a_{0..0,0..j}) = h_{pref}(a_{0..0,0..j-1}) + q^j a_{0j}$,

…,

$h_{pref}(a_{0..i,0..j}) = h_{pref}(a_{0..i-1,0..j}) + h_{pref}(a_{0..i,0..j-1}) - h_{pref}(a_{0..i-1,0..j-1}) + p^i q^j a_{ij}$.

Подібні поліноміальні хеші використовують, наприклад, у задачі пошуку найбільшої симетричної підматриці.

Поліноміальні хеші поширені у використанні для вирішення наступних задач:

1. Пошук всіх входжень одного рядку довжини n в інший довжини N з асимптотично мінімальним часом $O(n +$ N) (з перевіркою на істинність ще $O(n$ N)).

2. Пошук найбільшого спільного підрядку двох рядків довжин n и m ($n \geq m$) за $O((n+m\log(n))\log(m))$ та $O(n\log(m))$.
3. Знаходження лексикографічно мінімального циклічного зсуву рядку довжини n за $O(n\log(n))$.
4. Сортування всіх циклічних зсувів рядку довжини n у лексикографічному порядку за $O(n\log(n)^2)$.
5. Знаходження кількості підпаліндромів рядку довжини n за $O(n\log(n))$.
6. Кількість підрядків рядку довжини n, що є циклічними зсувами рядку довжини m, за $O((n+m)\log(n))$.
7. Кількість суфіксів рядку довжини n, нескінчене розширення яких співпадає з нескінченим розширенням всього рядку, за $O(n\log(n))$ (розширення – дублювання рядку нескінчену кількість разів).
8. Найбільший спільний префікс двох рядків довжини n з обміном двох довільних символів одного рядку місцями за $O(n\log(n))$.

Вказані асимптотичні оцінки наведені за умов, що у знаходженні хешу використовують сортування та бінарний пошук.

З наведеному переліку деякі із задач можуть бути розв'язані швидше іншими методами (наприклад, задача сортування циклічних зсувів, в якій шукати всі входження одного рядку в інший дозволяє алгоритм Кнута-Морріса-Пратта).

Лекція 4. Хешування зозулі та його модифікації. Фільтр Блума

Хешування зозулі – алгоритм уникнення колізій при утворені хеш-таблиць. Вид відкритої адресації, тому страждає від колізій [3]. Є одним із самих швидких способів хешування та має численні модифікації (класичний варіант алгоритму набагато швидший хешування з ланцюжками для малих хеш-таблиць, а блочний – і для великих).

Класичний варіант алгоритму. В алгоритмі використовують 2 хеш-функції $h_1(k)$, $h_2(k)$ з класу універсальних хеш-функцій, 1 хеш-таблицю (є модифікація, що використовує 2 хеш-таблиці, тобто по одній хеш-функції в одну хеш-таблицю):

$h_i(k) \in H_{pm}$, $i = \overline{1,2}$, $H_{pm} = \{ h_{ab} : a \in Z_p \ ma \ b \in Z_p^* \}$, $Z_p = \{0, 1, ..., p-1\}$, $Z_p^* = \{1, 2, ..., p-1\}$, $h_{ab} : Z_p^* \to Z_m$, $h_{ab}(k) = ((ak + b) \bmod p) \bmod m$.

Основні виконувані операції в хеш-таблиці: додавання елементу з ключем *k* до таблиці; вилучення елементу з ключем *k*; перевірка наявності в таблиці елементу з ключем *k*. Виконання вказаних функцій відбувається наступним чином.

Додавання елементу з ключем k до хеш-таблиці (add(*k*)).

1. Випадково обираємо комірку (корзину) $h_1(k)$ або $h_2(k)$ й дивимося чи вільна вона. Якщо ні, то перевіряємо другу комірку. Якщо одна з комірок з індексами $h_1(k)$ або $h_2(k)$ вільна – додаємо елемент з ключем *k* у хеш-таблицю.
2. Інакше довільно обираємо одну з цих комірок (корзин), запам'ятовуємо елемент, що в ній знаходиться, та розташовуємо новий елемент.
3. Звертаємося до комірки, на яку вказує інша хеш-функція від цього елементу з ключем, яку запам'ятали. Якщо вона вільна, то розташовуємо елемент у цій комірці.
4. Якщо ні, то запам'ятовуємо елемент з цієї комірки, розташовуємо там новий, перевіряємо чи не утворився цикл (якщо $h_1(k) = h_1(l) = h_1(z)$, $h_2(k) = h_2(l) = h_2(z)$).
5. Якщо все гаразд, то продовжуємо пошук вільного місця, доки не знайдемо його, або досягнемо ситуації зациклення, або досягнемо значення обмежувача на довжину ланцюга переміщень.
6. Якщо маємо зациклення, то обираємо дві нові хеш-функції та заново хешуємо усі додані елементи. Якщо ж досягли обмеження, то оголошуємо хеш-таблицю заповненою.
7. Після додавання потрібно збільшити розмір хеш-таблиці, якщо вона вже заповнена.

Вилучення елементу з ключем k з хеш-таблиці.

1. Дивимося у комірки з індексами $h_1(k)$, $h_2(k)$.
2. Якщо в одній з них є шуканий елемент, то помічаємо цю комірку як вільну.

Перевірка наявності елемента з ключем k у хеш-таблиці.

1. Дивимося у комірки з індексами $h_1(k)$, $h_2(k)$.

2. Якщо в одній з них є шуканий елемент, то повертаємо true, інакше false.

Звернемо увагу на можливість досягнення *зациклення при додаванні елементу*. Нехай додаємо елемент з ключом *k* у хеш-таблицю, а обидві комірки $h_1(k)$, $h_2(k)$ зайняті. Елемент з ключом *k* поклали у деяку комірку $h_i(k)$ хеш-таблиці. Якщо на наступному кроці знову бажаємо покласти елемент з ключом *k* у комірку $h_i(k)$, щоб у комірці $h_j(k)$, $i \neq j$, могли покласти деякий k_1, але якщо у ході переміщень *k* був переміщений до $h_j(k)$ (i = 1, 2, j = 1, 2), то вийде зациклення ($h_1(k) = h_1(l) = h_1(z)$, $h_2(k) = h_2(l) = h_2(z)$). Доведено, що уникнення проблеми зациклення дає зміна хеш-функції (з урахуванням обмежувача). Кількість ітерацій обмежують, уводячи обмежувач (задається наперед кількість ітерацій). Якщо число ітерацій досягнуто, то перефразують ключі у таблицях з використанням нових хеш-функцій та пробують ще раз, щоб пристосувати незайнятий ключ (або інакше збільшують розмір хеш-таблиці, що є свого роду зміною хеш-функції, так як розмір *m* враховується в обчисленні хеш-функції). Крім того, доведено, що з високим ступенем ймовірності для випадкового графа, у якому відношення ребра-вершини обмежене ($\leq$ ½), граф є псевдолісом, а тому алгоритм з успіхом розміщує всі ключі. І тому таке хешування буде ефективним за коефіцієнту заповнення таблиці $\alpha \leq$ ½. Для основних операцій асимптотичні оцінки часу роботи будуть наступними: для операцій вилучення та перевірки елементу з ключом *k* – $O(1)$ (потребує перегляду двох місць у хеш-таблиці, що вимагає константного часу у найгіршому випадку); для операції додавання елементу з ключом *k* до таблиці матимемо у середньому $O(1)$ (за урахування наявності обмежувача).

Лема. Додавання елементу з ключом *k* до хеш-таблиці у середньому відбувається за час $O(1)$.

Для доведення леми будується неорієнтований граф «зозулі», в якому кожній комірці хеш-таблиці відповідає одна вершина (усього *m* вершин), а кожному доданому елементу – ребро з кінцями у вершинах, що відповідають коміркам, у які вказують хеш-функції елементу. Елемент

буде доданий без перехешування тоді і тільки тоді, коли після додавання нового ребра граф залишається псевдолісом (тобто кожна його компонента зв'язності може містити не більше одного циклу). Звідси логічно випливають дослідження щодо випадкових блукань (випадкові польоти, блукання Леві тощо), тобто наскільки далеко можемо піти за фіксовану кількість ітерацій, але це вже перехід до розгляду алгоритмів безумовної оптимізації.

Використання двох хеш-функцій, що забезпечують два можливих положення в хеш-таблиці для кожного ключа є перевагою підходу (пари «ключ-значення» зберігаються у корзинах, що визначаються шляхом застосування хеш-функції до ключа). Подібні алгоритми можуть використовуватися у БД для індексації, у випадках, коли потрібен швидкий пошук.

Недоліком підходу є витрати пам'яті. Щоб гарантувати $O(1)$, потрібно, щоб пари «ключ-значення» займали не більше 50 % пам'яті ($\alpha \leq ½$). В алгоритмі послідовність зміщень може бути не дуже довгою, при досягненні обмежувача хеш-таблиця оголошується повною і потрібна буде зміна її розміру та перехешування. Крім того, додавання кожної нової хеш-таблиці суттєво збільшує середню швидкість заповнення таблиці (в БД під час перехешування немає відгуку).

Для випадку $\alpha > ½$ розглядають різноманітні модифікації алгоритму.

«Класики» (є модифікацією алгоритму «зозулі»). Основні виконувані функції в хеш-таблиці є такими ж. Відмінність є в операції додавання елементу з ключом *k* до хеш-таблиці. В даному алгоритмі можливі місця для вставки заданого ключа є суміжними. Кількість таких місцеположень називають *відстанню «стрибку»*. Під час вставки елемент хешується у визначене місцеположення на основі ключа. Потім він використовує лінійне пробування, щоб знайти пусту корзину. Якщо пуста корзина виходить за межі діапазону «стрибку», необхідно спробувати перемістити інші елементи з цього діапазону на місця, які вони можуть зайняти, вивільнивши при цьому місце для елементу, що вставляється. Якщо таке переміщення неможливе, хеш-таблиця стає повною й потрібно змінити її розмір з перехешуванням.

Алгоритм «Класики» дозволяє зберігати елемент поруч з корзиною, в яку він повинен попасти (задають деяке число H – максимальна відстань («стрибок»), на якій може розташовуватися елемент від своєї корзини. При вставці шукаємо вільну позицію для вставки, якщо вона

є в межах H, то вставляємо елемент, інакше, як було вказано, шукаємо елемент з іншої корзини, який можна посунути вправо в межах відстані H, і т.д., доки початковий елемент не попаде в межі H. Якщо це неможливо, то таблицю перехешовуємо. Для прискорення *пошуку* кожна корзина (комірка) зберігає бітову карту з H сусідніх елементів, у якій вказуються позиції в околі, які необхідно дослідити, щоб знайти елемент (відмічається, зайняте чи ні місто елементами з однаковим хеш-кодом). Таке хешування дозволяє уникнути проблеми кластерізації, покращує продуктивність при великій заповненості таблиці.

Отже, для найпростішого варіанту алгоритму «Класики» операція *додавання елементу з ключом k до хеш-таблиці* виглядатиме наступним чином:

1. Якщо комірка з індексом $h_1(k)$ вільна – додаємо елемент з ключом k у хеш-таблицю.
2. Інакше обираємо наступну комірку, якщо вона вільна, то додаємо з ключом k у хеш-таблицю, якщо ні, то звертаємося до наступних комірок з діапазону H, доки не зустрінемо вільну та розташовуємо новий елемент в ній.
3. Якщо в діапазоні H вільної комірки не зустріли, то розглядаємо комірку з індексом $h_2(k)$, якщо вона вільна, то додаємо елемент з ключом k у хеш-таблицю, інакше послідовно звертаємося до наступних комірок з діапазону H, доки не зустрінемо вільну та розташовуємо новий елемент в ній.
4. Якщо не зустріли жодної вільної комірки, то оголошуємо хеш-таблицю заповненою та перехешовуємо її.

Хешування «зозулі» та «класики» не вимагають динамічного розподілу пам'яті й підтримують доволі високий коефіцієнт завантаження з обмеженим часом пошуку, але ці хеш-таблиці страждають від *каскадних записів* (множинності операцій запису для однієї вставки).

«Зозулине» хешування із «запасом» («запас» – масив ключів сталої довжини, що використовують для збереження ключів, які не можуть бути успішно встановлені у головну хеш-таблицю, тобто – це додаткова таблиця) використовує три хеш-функції: дві для головної хеш-таблиці та одну для додаткової. Використання «запасу» дозволяє уникнути циклів.

Для *додавання елементу з ключом k до хеш-таблиці при хешуванні із «запасом»* можлива наступна послідовність дій:

1. Якщо одна з комірок з індексами $h_1(k)$ або $h_2(k)$ вільна – додаємо елемент з ключом *k* у хеш-таблицю.
2. Інакше довільно обираємо одну з цих комірок (корзин), запам'ятовуємо елемент, що в ній знаходиться та розташовуємо новий елемент.
3. Звертаємося до комірки, на яку вказує інша хеш-функція від цього елементу з ключом, яку запам'ятали. Якщо вона вільна, то розташовуємо елемент у цій комірці.
4. Якщо ні, то звертаємося до запасу та розташовуємо там елемент у комірці, на яку вказує $h_3(k)$.
5. Після додавання потрібно збільшити розмір хеш-таблиці, якщо вона оголошена вже заповненою.

Вилучення елементу з ключом k з хеш-таблиці при хешуванні із «запасом».

1. Дивимося у комірки з індексами $h_1(k)$, $h_2(k)$.
2. Якщо в одній з них є шуканий елемент, то помічаємо цю комірку як вільну.
3. Якщо він відсутній, то звертаємося до «запасу» у комірку з індексом $h_3(k)$ і там помічаємо комірку як вільну.

Перевірка наявності елемента з ключом k у хеш-таблиці при хешуванні із «запасом».

1. Дивимося у комірки з індексами $h_1(k)$, $h_2(k)$.
2. Якщо в одній з них є шуканий елемент, то повертаємо true.
3. Інакше звертаємося до «запасу» у комірку з індексом $h_3(k)$, якщо шуканий елемент там, то повертаємо true, інакше false.

Модифікація алгоритму «зозулі» із «запасом» сильно зменшує число невдач шляхом за збільшення розміру запасу. Але більший запас означає повільний пошук ключа, якого немає в основній таблиці або, якщо він знаходиться в запасі. Для різноманітних модифікацій алгоритму «зозулі» із «запасом» загальновживаним є правило: *запас не може бути використаний з більше ніж двома хеш-функціями* або з блочним зозулиним хешуванням. У загальному використання алгоритму дає

підвищення степені завантаження хеш-таблиці та зменшення числа невдач вставки.

PCM Friendly HashTable (PFHT) є модифікацією алгоритму «зозулі». Побудований для створення ефективно працюючої з пам'яттю PCM хеш-таблиці, що не страждає від збільшення часу пошуку, дає переваги алгоритму «зозулі» (як багаторівневої побудови), та зберігає ефективне використання пам'яті при високому коефіцієнті заповнення таблиці, дає гарантований час пошуку значення, уникає проблем каскадних записів, що впливають на продуктивність. В алгоритмі використовують основну таблицю з двома хеш-функціями та «схованку» із своєю хеш-функцією, що допомагає досягти високого коефіцієнту заповнення. Основна таблиця реалізується як двовимірний масив корзин. Корзини зберігаються у пам'яті неперервно. Кожна корзина містить фіксовану кількість місць для зберігання пар «ключ-значення». Алгоритм PFHT використовує 2 варіанти для зберігання будь-якої вхідної пари. Крім того, кожна корзина заповнюється однією або декількома послідовними лініями кешу. Тому кількість місць у корзині залежить від розміру рядку кешу, що є доволі великим у сучасних процесорах. У «схованці» зберігають будь-які пари, які не можна було вставити в основну таблицю, підвищуючи при цьому коефіцієнт заповнення (тут використовується третя хеш-функція $h_3(k)$). Перевагою алгоритму є *збалансована* вставка (пара може зберігатися в одній з двох корзин у головній таблиці, $h_1(k)$, $h_2(k)$, обирається найменш завантажена корзина (в хешуванні «зозулі» корзина обирається випадково), відсутність каскадних записів (для цього даний алгоритм дозволяє тільки одне переміщення, під час вставки, якщо обидві корзини зайняті, алгоритм перевіряє, чи можна перенести будь-яку пару на зайнятому місці в альтернативне, якщо ні, то пара, що вставляється розташовується у «схованці»). Алгоритм дозволяє зберігати ключі в одному місці у суміжних корзинах, що покращує продуктивність кешу. Для досягнення високого коефіцієнту заповнення алгоритм використовує великий розмір корзин (але це збільшує час очікування для пошуку). Емпірично встановленим значенням розміру корзини тут є розмір у 2 рядки кешу.

Блочне хешування – спосіб вирішення колізій, де блоки є масивами фіксованої довжини. В модифікованих алгоритмах «зозулі» також використовують блочне хешування. В модифікаціях, що використовують блочне хешування, виділяють ряд блоків, кожен з яких може містити

декілька елементів (більше 1 ключа на комірку). Для вставки елемента у таблицю він відображується на один з блоків і потім розміщується у цей блок (якщо блок заповнений, то виконується обробка переповнення). Такий підхід підвищує відсоток завантаження до 80% (але це вже криптографічне хешування).

В модифікованих алгоритмах «зозулі» може використовуватися декілька хеш-таблиць (3 хеш-таблиці або 2). Якщо використовують *дві* хеш-таблиці, то хеш-таблицю поділяють на дві меншого розміру й кожна з хеш-функцій дає індекс лише в одну з цих таблиць.

В модифікованих алгоритмах «зозулі» також можливе використання більше двох основних хеш-функцій. Використання *трьох* хеш-функцій підвищує відсоток завантаження до 91% (результат встановлений емпірично).

Фільтр Блума – це структура даних для перевірки належності елемента до множини (наприклад, якоїсь небажаної інформації). На практиці фільтр Блума застосовують для фільтрації запитів до зовнішньої пам'яті або даних, що передаються по мережі, щоб уникнути високовартісних операцій доступу, у проксісерверах, БД і т.д. У фільтрі Блума можливі *хибнопозитивні* спрацювання, але не хибнонегативні.

Зозулін фільтр, що є структурою даних з нечіткими множинами, що базується на хешуванні «зозулі» дозволяє і вилучати елемент, а класичний фільтр Блума – тільки вставку й перевірку належності.

Фільтр Блума містить масив з m біт та декілька хеш-функцій $h_0, h_1, h_2, ..., h_{l-1}$, що належать класу універсальних хеш-функцій H_{pm} та є незалежними [3]. Коли фільтр не містить жодного елемента, то всі m бітів є нульовими. При додаванні елементу з ключом k обчислюються позиції $h_0(k), h_1(k), h_2(k), ..., h_{l-1}(k)$. Елемент з ключом k можливо міститься у фільтрі, тільки якщо на всіх цих позиціях у масиві стоять 1, інакше елементу немає в множині (що задається фільтром). Така структура звичайно займає менший об'єм пам'яті, ніж хеш-таблиця, але не є детермінованою множиною.

Чим більший об'єм пам'яті, що є наперед заданим нами, тим менша ймовірність хибного спрацювання. Недоліком є те, що у класичному варіанті підтримуються лише операції додавання та перевірки наявності (пошуку не потрібно), а вилучення немає (для уникнення цього існують різні модифікації з лічильниками).

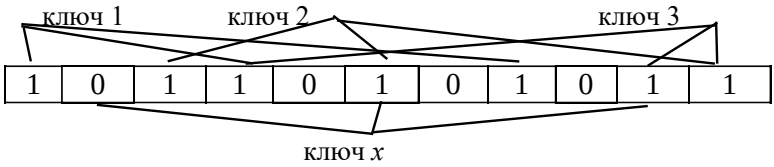


Рис. 4.1. Перевірка наявності ключа у фільтрі (три хеш-функції)

Хеш-функції $h_0, h_1, h_2, ..., h_{l-1}$ рівноймовірно відображають елементи початкової множини у множину $\{0,1,..., m-1\}$ відповідним номером бітів у масиві (результат їх може бути 0,1,…,*m*-1). Для додавання елементу *x* до множини необхідно записати 1-ці на кожну з позицій $h_0(x), h_1(x), h_2(x), ..., h_{l-1}(x)$ бітового масиву. Щоб перевірити, чи елемент *x* належить множині елементів, що зберігаються у фільтрі, необхідно перевірити стани бітів $h_0(x), h_1(x), h_2(x), ..., h_{l-1}(x)$ (якщо l = 3 як на рис. 4.1, перевіряємо ці 3 біти). Якщо всі вони дорівнюють 1, структура відповідає, що елемент належить множині (але може бути хибне спрацювання). Оптимальний розмір масиву в бітах:

$$m = \frac{-n \ln p}{(\ln 2)^2}, \qquad (4.1)$$

n – припустима кількість елементів, що зберігаються у фільтрі-множині, *p* – бажана ймовірність хибного спрацювання.
Оптимальна кількість хеш-функцій:

$$l = \frac{m}{n} \ln 2 \text{, що дає також } l = -\frac{\ln p}{\ln 2}. \qquad (4.2)$$

При роботі фільтру, як вже вказувалося, перевіряється наявність елементу в множині, але він не дістається, тому в фільтрі не витрачається час на пошук. Миттєва перевірка – чи є деякі дані, наприклад, спам, шкідливі файли. Інформацію розбивають на блоки та додають у фільтр. Коли потрібно просканувати дані на наявність шкідливої активності, то беруть блоки сховища даних та перевіряють чи є вони у фільтрі.
Основні операції у фільтрі: додавання елементу до множини, що представлена фільтром; перевірка належності елементу до множини, що представлена фільтром. Виконання вказаних функцій відбувається наступним чином.

Додавання до фільтру [3]: InsBlF(T,x)
Вхід: T – бітовий масив розміром m,
x – запис на додавання до множини, що представлена фільтром
Вихід: запис доданий до фільтру, тобто встановлені біти $h_0(x), h_1(x), h_2(x), ..., h_{l-1}(x)$ на 1

for i = 0..l-1
обраховуємо $h \leftarrow h_i(x)$
встановлюємо $T(h) \leftarrow 1$

Перевірка [3]: ScinBlF(T,x)
Вхід: T – бітовий масив розміром m,
x – запис на перевірку належності до множини, що представлена фільтром
Вихід: true, false

for i = 0..l-1
обраховуємо $h \leftarrow h_i(x)$
if $T(h) = 0$ then
return false
return true

Розглянемо як отримати формули для оптимальних значень (4.1), (4.2). Якщо є одна хеш-функція, то ймовірність, що конкретний біт у таблиці буде розміщений, коли ми хешуємо елемент дорівнює $1/m$. Тому ймовірність того, що конкретний біт не буде встановлений буде $1 - 1/m$. Якщо маємо l незалежних хеш-функцій, то ймовірність, що конкретний біт не буде розміщений дорівнює $(1 - 1/m)^l = (1 - 1/m)^{ml/m} = (1/e)^{l/m}$. Якщо у фільтрі n елементів, то ймовірність того, що біт не буде розміщений дорівнює $p = (1/e)^{ln/m}$, а ймовірність того, що біт буде розміщений дорівнюватиме $1 - e^{-ln/m}$. Тоді ймовірність одержання хибнопозивного результату дорівнює ймовірності, що після додання n елементів перевіряємо елемент, який не у фільтрі, а всі l хеш-значень цього елементу знаходяться у зайнятих позиціях (дорівнює ймовірності, що l конкретних бітів розміщені) $p = (1 - e^{-ln/m})^l$. Звідси і одержуються оптимальні значення для фільтру Блума: $m = \frac{-n \ln p}{(\ln 2)^2}$ та $l = -\frac{\ln p}{\ln 2}$.

Нехай $m = 1 \cdot 10^{10}$ біт, $n = 1 \cdot 10^{9}$ елементів у фільтрі. Тоді $l = \frac{m}{n} \ln 2 \approx 7$ різних незалежних хеш-функцій, а ймовірність хибнопозитивного результату буде: $p = (1 - e^{-7/10})^7 \approx 0.008 < 1\%$. Проте навіть з 5 хеш-функціями можемо досягти менше 1% для хибнопозитивного результату.

Якщо маємо значення l, задана ймовірність p та кількість елементів n, то визначаємо кількість біт для фільтра. За $p = 1\%$, $n = 1 \cdot 10^{9}$ елементів, $l = 7$, то $m \approx 1 \cdot 10^{10}$ біт. Якщо середній розмір елемента складає 10 біт, то можемо опрацювати більше 10 Гбайт даних з обчислювальною швидкістю декількох хеш-функцій.

Можливе об'єднання, перетин та ієрархічна побудова фільтрів Блума. Перетин та об'єднання з однаковим розміром і набором хеш-функцій реалізують на практиці за допомогою відповідно операцій порозрядного ТА й АБО. Операція об'єднання фільтрів не має втрат в тому смислі, що результуючий фільтр Блума співпадає з фільтром Блума, просто створеним з використанням об'єднання двох наборів. Операція перетину має наступну властивість: ймовірність хибного спрацювання у результуючому фільтрі Блума не більш чим ймовірність хибного спрацювання в одному із складових фільтрів, проте може бути більшою, ніж ймовірність хибного спрацювання у фільтрі, просто створеного на основі перетину відповідних двох наборів.

Існує багато модифікацій фільтру Блума: підрахункові, розподілені, реплікаційні, фільтри Блумье, стабільні фільтри Блума, масштабовані, багатошарові, послаблені та ін.

Лекція 5. Циклічні надлишкові коди (CRC). Прямі алгоритми CRC

Розглянемо некриптографічні CRC, які застосовуються як хеш-функції та як перешкодостійке кодування. Некриптографічні CRC використовують у різних протоколах зв'язку для перевірки цілісності даних, що передаються різними пристроями, при випадкових викривленнях інформації в каналі передачі даних, але не використовують для захисту [4]. Алгоритми таких CRC мають невисокі витрати ресурсів та прості у

реалізації. Часто реалізуються на апаратному рівні (архіватор стискує файли у відповідності з деяким алгоритмом архівації, обчислюючи з кожного стиснутого файлу CRC, потім заархівовані файли можна копіювати, пересилати тощо, а інформація може викривитися, навіть у 1 біті; коли архів розпаковують, архіватор у першу чергу перевіряє цілісність файлів (знову обчислює CRC та порівнює з першим значенням CRC, якщо вони рівні, значить цілісність порушена не була та архів розпаковують, інакше – ні)). CRC не обов'язково обчислювати для великих масивів даних (файл), можна це робити для окремих рядків, слів тощо. Для реалізації CRC використовують теорію лінійних циклічних кодів.

Важливим поняттям є поняття **контрольної суми** (КС) – деякого значення, обчисленого за визначеною схемою на основі кодового повідомлення. Використовують в якості перевірочної інформації, що дописується до даних, які передаються. На приймаючій стороні відомий алгоритм обчислення КС і відбувається перевірка прийнятих даних.

Сутність таких дій: за добрих характеристик КС похибка у повідомленні переважно приведе до зміни у КС. Якщо КС початкова та перевірочна не однакові, то приймається рішення про недостовірність прийнятих даних, відбувається запит на повторну передачу пакету. Визначення поодиноких похибок відбувається з приблизно 100 % ймовірністю, інших – з ймовірністю $1 - 2^{-N}$, де N – число розрядів контрольного коду.

Алгоритми CRC базуються на поліноміальній арифметиці: повідомлення, дільник та залишок мають бути представлені у вигляді поліномів з двійковим коефіцієнтами (послідовності бітів, кожен з яких є коефіцієнтом поліному).

Алгоритм CRC використовує властивості ділення із залишком двійкових багаточленів (значення CRC є залишком від ділення багаточлена, що відповідає вхідним даним, на деякий фіксований *породжуючий багаточлен*).

Для кожної послідовності бітів $a_0 \ldots a_{N-1}$ взаємно однозначно складається двійковий поліном $a_0x^{N-1}+\ldots+a_{N-1}$, послідовність коефіцієнтів якого являє собою вихідну послідовність. Послідовність бітів 1011010 відповідає багаточлену $P(x) = x^6+x^4+x^3+x$.

Кількість різних багаточленів степені менше N дорівнює 2^N, що співпадає з числом усіх двійкових послідовностей довжини N.

Значення КС визначають як бітову послідовність довжини N, що є багаточленом R(x), який є залишком від ділення багаточлена P(x), що є вхідним потоком біт, на багаточлен G(x): $R(x)=P(x)x^N \mathrm{mod} G(x)$, де G($x$) – породжуючий багаточлен степені N.

В CRC алгоритмі використовують поліноміальну арифметику за модулем 2, тому дії, що виконуються під час обчислення CRC є арифметичними операціями без урахування переносу. Операції складання та віднімання тут ідентичні операції XOR (виключаюче АБО). Ділення відбувається аналогічно до звичайного, але замість віднімання при діленні стовпчиком виконується операція XOR. На рис. 5.1. представлене виконання операції над повідомленням 1101011011 з поліномом 10011. Отриманою контрольною сумою (CRC) є 1110, що відповідає залишку від ділення без урахування переносу (а результат, тобто частка, не потрібний для використання).

```
                    1100001010   частка
10011       11010110110000   вирівняне повідомлення (1101011011+0000)
Поліном     10011
             10011
             10011
                 1
              00000
                 10
               00000
                 101
                00000
                 1011
                00000
                 10110
                 10011
                  01010
                  00000
                   10100
                   10011
                      1110   залишок (контрольна сума)
```

Рис. 5.1. Обчислення побітово контрольної суми (КС).

Часто з файлу береться перше слово (для CRC-1 маємо бітовий елемент, для CRC-8 – байтовий), у найпростішому алгоритмі, якщо старший біт у слові «1», то слово зсувають вліво на 1 розряд з наступним виконанням операції XOR з породжуючим поліномом. Якщо старший біт «0», то зсувають вліво на 1 розряд без виконання операції XOR. Так як результат не представляє жодного інтересу, то після зсуву старший біт втрачається. На місце молодшого біту (яке тепер містить «0») загружають наступний біт з файлу і т.д. до останнього біту з файлу. Після проходження усього повідомлення у схові залишається залишок – контрольна сума (КС). Якщо, наприклад, потрібно отдержати контрольну суму 8-розрядну, то утворюючий поліном береться 9-розрядний (тобто має 8-му степінь).

Множення на x^N – це приписування N нулевих бітів до вхідної послідовності, що покращує якість хешування для коротких вхідних послідовностей (можемо одержати 2^N різних залишків від ділення).

Багаточлен $G(x)$ часто є непривідним. Звичайно його підбирають у відповідності з вимогами до хеш-функції для кожного конкретного застосування (існують стандартні, що можуть задати мінімальне число колізій, просто обчислюватися). Але стандартно використовувані поліноми не є самими ефективними.

Важливою характеристикою є степінь поліному (ширина – width). Часто вибирають величини, кратні до розрядності регістрів процесору для спрощення реалізації алгоритмів (16, 32 та ін.). Степінь поліному – це дійсна позиція старшого біту. Позиції бітів відліковують починаючи з 0. Отже, породжуючий поліном, а саме його степінь, визначає кількість бітів, використовуваних для обчислення CRC. Ще також важливим параметром є: початкове (стартове) значення слова.

Існують різні модифікації алгоритму (прямий, дзеркальний, табличний (що використовує асоціативність операції XOR) тощо).

Найпростіший прямий **алгоритм** – **побітовий зсув** [5], який можна представити у наступному вигляді.

1. Створюється масив (регістр), заповнений 00…00, що дорівнює за довжиною розрядності (степені) поліному.
2. Початкове повідомлення заповнюється 00…00 у молодших розрядах, за кількістю рівних числу розрядів поліному.

3. У молодший розряд регістру заносять старший біт повідомлення, а із старшого розряду регістру висувають один біт.
4. Якщо висунутий біт дорівнює 1, то відбувається інверсія бітів у тих розрядах регістру, що відповідають першим в поліномі.
5. Якщо у повідомленні є ще біти, то переходять до виконання 3.
6. Якщо всі біти повідомлення оброблені алгоритмом, у регістрі залишається залишок від ділення, який і є CRC (КС).

Отже, повідомлення T, що передається, є кратним дільнику (останні W бітів повідомлення T – це залишок від ділення повідомлення M на дільник: T = M + CRC). А так як додавання є одночасно й відніманням, то воно зменшує M до найближчого кратного.
Оскільки повідомлення може бути дуже великим (наприклад, декілька Мбт) та використовується для отримання контрольної суми CRC-арифметика, то використання звичайної комп'ютерної операції ділення неможливе. Найпростішим підходом є наступний. Нехай поліном з N = 4 має вигляд 10111. Тоді регістр буде чотирьохбітним, а алгоритм, що повертає значення регістру, де буде знаходитися КС, буде такий:

```
CRC_простий(КС)
Завантажити регістр нульовими бітами
Доповнити кінцеву частину повідомлення N нульовими бітами
while (є необроблені біти)
    begin
    зсунути регістр на 1 біт вліво
    розмістити черговий ще не оброблений біт повідомлення у 0 позицію регістру
    if (з регістру був висунутий біт із значенням «1»)
      регістр = регістр XOR поліном
end
return регістр
```

Якщо ж до регістру на початку обчислень записати не 00…00, а F…F, то це підвищить надійність визначення початку передачі повідомлення, якщо, наприклад, повідомлення на початку має нульові біти. Крім того, необхідно відокремлювати повідомлення та значення КС. Уникнути прямого відслідковування довжини рядку, що приймається, можна шляхом формування джерелом КС на початку повідомлення або, взагалі, окремо від рядку (це характерно для програми, що відслідковує цілісність файлів у деякому каталозі, тоді результати CRC для кожного

файлу зберігаються у деякому службовому файлі). В деяких модифікаціях алгоритму може виконуватися й таке: - безпосередньо перед видачою КС результат ділиться на яке-небудь інше число; - байти повідомлення у процесі запису до регістру можуть розміщуватися як старшим бітом уперед, так і молодшим; - результуючий CRC може також видаватися, починаючи від старшого біту або від молодшого (реверс).
Тому на значення КС впливають наступні параметри:

- порядок CRC;
- утворюючий багаточлен;
- початковий вміст регістру;
- значення, з якого відбувається фінальний XOR;
- реверс байтів повідомлення;
- реверс байтів CRC перед фінальним XOR.

Але, взагалі, для прискорення можна виконувати ділення й так:

```
101111001110 |10011
10011
…10010
  10011
  00001 0111
      1 0011
        1000  – КС
```

Хоча й цей підхід, внаслідок дуже великої довжини повідомлень, вважають повільним. Вдосконалення такого підходу до ділення привело до появи табличних алгоритмів пошуку CRC.

Прямий табличний алгоритм. Отже, побітовий алгоритм для практичних повідомлень є надто повільним та неефективним. Асоціативність операції XOR дозволяє проводити обчислення з цілими байтами. Представимо такі обчислення, наприклад, для поліному 32 степені (N = 32).

Розглянемо регістр, який використовують для збереження проміжного результату обчислення CRC. Коли біт, що висунутий з лівої частини регістру, є 1, то виконується операція XOR вмісту регістру з молодшими N бітами поліному (тут N = 32), тобто виконуємо ділення подібно до наведеного вище. Якщо ж зсувати одночасно цілі групи біт, то можемо мати наступне.

Нехай до зсуву у регістрі маємо: 10110100110000010001000100010001. Потім 16 біт висуваються зліва, а справа надходять нові 16 біт (наприклад, 0010001000100011). Тоді поточний вміст регістру: 00010001000100010010001000100011. Біти, що тільки но висунуті зліва: 1011010011000001. Поліном, наприклад, заданий такий: 10100000000000000000000000010101111.

Обчислимо нове значення регістру:

10110100110000010001000100010001000100011

10100000000000000000000000010101111

Поліном складаємо по XOR починаючи з 15 біту старшої групи. Получимо:

00010100110000010001000101000110 1.

Далі поліном складаємо по XOR знову.

10100110000010001000101000110101000100011

10100000000000000000000000010101111

Маємо

00000110000010001000101001100010100100011

Загалом старші 8 бітів тепер містять нулі. Теж саме могли б отримати, якщо скласти операнди у першому та другому діленні (поліноми), що прикладають до відповідних бітів старшої групи (0 та 3), й результат застосувати по XOR до початкового значення регістру. Це вказує на можливість наперед розрахувати величини XOR-доданку для кожної комбінації старших бітів.

Якби працювали просто послідовно, то мали б наступне.

10100000000000000000000000010101111

Результат

01100001000100010100110000000 1011

Знову складаємо по XOR.

11000010001000101001100000001011 0100011

10100000000000000000000000010101111

Результат

01100010001000101001100001011100 1

Знову

11000100010001010011000010111001100011

10100000000000000000000000010101111

Результат

01100100010001010011000011101 1100

Знову
11001000100010100110000111011100 00011
10100000000000000000000000010101111
Результат
01101000100010100110000110001011 1
Знову
11010001000101001100001100010111 0011
10100000000000000000000000010101111
Результат
01110001000101001100001101000000 1
Знову
11100010001010011000011010000001 011
10100000000000000000000000010101111
Результат
01000010001010011000011011010110 1
Знову
10000100010100110000110110101101 11
10100000000000000000000000010101111
Результат
00100100010100110000110111111010 0
Знову
10010001010011000011011111101001
10100000000000000000000000010101111
Але вже не вистачає біта для ділення. Якщо немає необроблених бітів, то 10010001010011000011011111101001 є CRC, якщо є ще необроблені біти повідомлення то зсуваємо біти вліво й до регістру справа надходять нові біти (32). Але це не ефективно.
Якщо ж для кожної комбінації бітів старшої групи, що висуваються з регістру можемо розрахувати доданок, то одержимо відповідну таблицю (якщо 8 бітів, то з 256 подвійних слів).
Отже, алгоритм можна оптимізувати. Немає потреба постійно протискати байти рядку через регістр. Можна безпосередньо застосовувати операцію XOR байтів, висунених з регістру, з байтами оброблюваного рядку. Результат буде вказувати на позицію в таблиці, значення з якої й буде на наступному кроці складатися з регістром. Обчислення таблиці відбувається одноразово, результати зберігаються.

Тоді, наприклад, для N = 8 можемо одержати такий найпростіший варіант алгоритму.

Прямий табличний алгоритм [5].

1. Вибираємо перший байт масиву, який розглядається як адреса в таблиці. Вибираємо в таблиці число з даним номером (адресою) – одержуємо перший залишок.
2. Беремо другий байт інформаційного масиву та сумуємо його за mod2 із залишком. Одержане число є адресою в таблиці. Для цієї адреси беремо з таблиці залишок два.
3. Далі беремо наступний байт масиву, сумуємо за mod2 із залишком два. Одержане число є адресою в таблиці. Для цієї адреси беремо з таблиці залишок.
4. Виконуємо 3 доки не досягнемо кінця масиву, після його обробки у вихідному регістрі буде знаходитися значення КС.

Або більш докладно:

While (повідомлення повністю не оброблене)
Begin
Перевіримо старший байт регістру
Розрахуємо на його основі контрольний байт
Базуючись на значенні контрольного байту,
виконаємо операцію XOR обраного поліному з різними зміщеннями
Зсунемо регістр на один байт вліво, додавши справа новий байт із повідомлення
Виконаємо операцію XOR вмісту регістру з сумованим поліномом
End

Поки що наведений алгоритм не є кращим за "простий" прямий алгоритм. Але якщо частину обчислень, як було вказано вище, провести наперед, а результати зібрати у таблицю, тоді маємо:

While (повідомлення повністю не оброблене)
Begin
Top = Старший_байт(Register);
Register = (зсув Register на 8(16…) біт вліво) **або** Новий_байт_повідомлення;
Register = Register XOR Розрахована_таблиця[Top];
End

Отриманий алгоритм вже став ефективним алгоритмом, він вимагає тільки операцій зсуву, OR, XOR, а також операції пошуку в таблиці (не враховуючи попередні обчислення в таблиці). Графічно схема алгоритму виглядає як на рис. 5.2.

Отже, операції зсуву на 1 байт, доповнення регістру справа новим байтом з повідомлення, XOR табличного значення із вмістом регістру виконуються за константний час кожна. Залишаються ще операції пошуку у наперед сформованій таблиці.

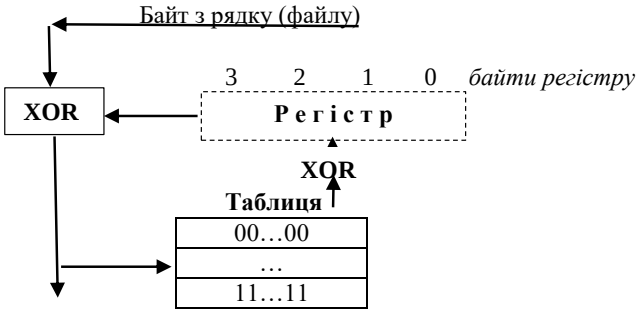


Рис. 5.2. Схема алгоритму [5].

Тому розглянемо як формують таблицю. Розмір таблиці залежить від розрядності КС. Наприклад, маємо утворюючий поліном восьмого порядку, тобто розрядність КС – $N = 8$.

Складається таблиця чисел $2^N \times N$ (залишків від ділення на поліном), тому для восьмирозрядної КС матимемо таке:

Адреса в таблиці	Дані в таблиці, ділення за mod 2
1	Залишок від ділення числа 10000000 на поліном
10	Залишок від ділення числа 100000000 на поліном
11	Залишок від ділення числа 110000000 на поліном
…..	……
1111 1111	Залишок від ділення числа 111111110000000 на поліном

Звідси бачимо, недоліком є те, що для КС з великим числом розрядів буде потрібно великий обсяг пам'яті. Тому вважають доцільним використання таблиць при $N \leq 16$.

В результаті на практиці, наприклад, для **CRC-16** застосовують подібний *алгоритм*:

1. Висувається старший байт регістру у байтову комірку.
2. Виконується XOR над висунутим байтом регістру та байтом рядку-повідомлення.
3. Одержане значення є індексом для доступу до наперед створеної таблиці.
4. Узяте з таблиці значення об'єднується по XOR із значенням у регістрі та результат заноситься у регістр.
5. До закінчення обробки останнього байту рядку виконуються кроки 3-4, після обробки останнього байту рядку у регістрі буде міститися CRC.

Лекція 6. Обернені алгоритми CRC. Аспекти побудови практичних алгоритмів CRC

Дзеркальні CRC алгоритми [5]. Найпростіший з них – послідовний, в якому надсилаються біти починаючи найменшого і закінчуючи найстаршим, тобто у зворотному порядку. У «дзеркальному» регістрі всі біти відображаються відносно центру. Замість того, щоб міняти місцями біти перед їх обробкою, дзеркально відображають всі значення та дії, що беруть участь у прямому алгоритмі. При цьому напрямок розрахунків змінюється – байти зсуваються направо, а також додається дзеркальне відображення поліному (рис. 6.1). Для дзеркального табличного алгоритму таблиця стане дзеркальним відображенням CRC-таблиці прямого алгоритму.

Отже, для дзеркального послідовного алгоритму (наприклад, CRC-32) *процес обчислення аналогічний, але інший порядок зсувів та заповнення регістру (заповнюється вправо)*. Проміжний CRC для першої дії може бути 16 бітів. Байти рядку розглядаються *без дописування нулів* у кінці. Дзеркальний табличний алгоритм (CRC-32) може мати вигляд:

1. Заповнюємо регістр та висувається старший байт регістру у байтову комірку (на 1 байт вправо).
2. Виконується XOR над висунутим байтом регістру та правим байтом рядку-повідомлення.
3. Одержане значення є індексом для доступу до наперед створеної таблиці.
4. Узяте з таблиці значення об'єднується по XOR із значенням у регістрі та результат заноситься у регістр.
5. До закінчення обробки останнього байту рядку виконуються кроки 3-4, після обробки останнього байту рядку у регістрі буде міститися CRC.

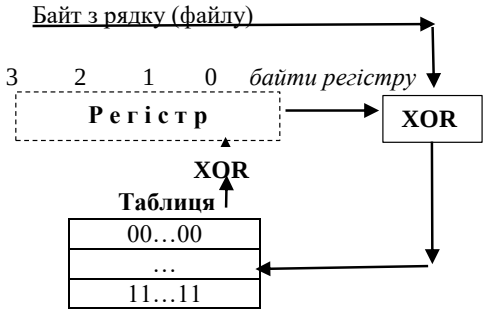


Рис. 6.1. Схема оберненого табличного алгоритму.

Для прямих та дзеркальних алгоритмів на практиці використовують також різні додаткові ускладнення. Наприклад, можливе повернення старших бітів й перемішування з молодшими, додавання віртуального байту, помноженого на деяку емпіричну сталу.

При виборі конкретного CRC алгоритму важливим моментом є врахування типів помилок, що очікувані при передачі повідомлень. Найпростішими типами помилок, які найбільш очікувані при передачі є [5]:

- *Помилка в одному біті.* E = 100...000. Такі помилки можна відловити якщо у поліномі не менше 2-х бітів встановлено в 1. Як би не складали зсуви таких поліномів, завжди мали б рядок,

в якому хоча б 2 одиничних біта, а значить E не може бути кратним поліному.

- *Помилка в двох бітах.* Для усунення таких помилок підбирають поліноми, для яких не є кратними числа типу 011, 0101, 01001 і т.д.
- *Помилка у непарній кількості біт.* Такі похибки можна відловити, якщо взяти поліном, в якому кількість біт парна.
- *Помилка пакета.* E = 000...000111...111000...000. Тобто похибка локалізована у неперервному пакеті у повідомленні. Таке E можна представити у виді добутку: E=(1000.....00)*(111111111). Якщо в поліномі молодший біт «1», то лівий множник не може бути дільником поліному, тоді, якщо поліном довший за правий множник, то похибка буде відловлена.

Розроблені для практичного застосування CRC алгоритми використовують наступні стандартизовані специфікації, що встановлюють повну характеристику конкретного алгоритму (табл. 5.1):

- Назва алгоритму;
- Степінь породжуючого КС багаточлена (width);
- Породжуючий поліном. Його спочатку записують як бітову послідовність (при цьому старший біт випускають – він завжди 1). Наприклад, багаточлен буде записаний числом . Для зручності двійкове представлення записують у шістнадковій формі (або 0x11);
- Стартові дані (init), тобто значення регістрів на момент початку обчислень;
- Прапорець (RefIn), що вказує на початок і напрямок обчислень, для виявлення пакетів помилок має відповідати порядку передачі у каналі. Існує два варіанти: False —починаючи із старшого значущого біту, True — з молодшого;
- Прапорець (RefOut), вказує чи інвертується порядок бітів регістру при вході на елемент XOR;
- Число (XorOut), з яким складається за mod2 одержаний результат;
- Значення CRC (check) для рядку «123456789».

Табл. 5.1. Специфікації поширених CRC алгоритмів.

Назва	Width	Поліном	Init	RefIn	RefOut	XorOut	Check
CRC-3/ROHC	3	0x3	0x7	true	true	0x0	0x6
CRC-4/ITU	4	0x3	0x0	true	true	0x0	0x7
CRC-5/EPC	5	0x9	0x9	false	false	0x0	0x0
CRC-5/ITU	5	0x15	0x0	true	true	0x0	0x7
CRC-5/USB	5	0x5	0x1F	true	true	0x1F	0x19
CRC-6/CDMA2000-A	6	0x27	0x3F	false	false	0x0	0xD
CRC-6/CDMA2000-B	6	0x7	0x3F	false	false	0x0	0x3B
CRC-6/DARC	6	0x19	0x0	true	true	0x0	0x26
CRC-6/ITU	6	0x3	0x0	true	true	0x0	0x6
CRC-7	7	0x9	0x0	false	false	0x0	0x75
CRC-7/ROHC	7	0x4F	0x7F	true	true	0x0	0x53
CRC-8	8	0x7	0x0	false	false	0x0	0xF4
CRC-8/CDMA2000	8	0x9B	0xFF	false	false	0x0	0xDA
CRC-8/DARC	8	0x39	0x0	true	true	0x0	0x15
CRC-8/DVB-S2	8	0xD5	0x0	false	false	0x0	0xBC
CRC-8/EBU	8	0x1D	0xFF	true	true	0x0	0x97
CRC-8/I-CODE	8	0x1D	0xFD	false	false	0x0	0x7E
CRC-8/ITU	8	0x7	0x0	false	false	0x55	0xA1
CRC-8/MAXIM	8	0x31	0x0	true	true	0x0	0xA1
CRC-8/ROHC	8	0x7	0xFF	true	true	0x0	0xD0
CRC-8/WCDMA	8	0x9B	0x0	true	true	0x0	0x25
CRC-10	10	0x233	0x0	false	false	0x0	0x199
CRC-10/CDMA2000	10	0x3D9	0x3FF	false	false	0x0	0x233
CRC-11	11	0x385	0x1A	false	false	0x0	0x5A3
CRC-12/3GPP	12	0x80F	0x0	false	true	0x0	0xDAF
CRC-12/CDMA2000	12	0xF13	0xFFF	false	false	0x0	0xD4D
CRC-12/DECT	12	0x80F	0x0	false	false	0x0	0xF5B
CRC-13/BBC	13	0x1CF5	0x0	false	false	0x0	0x4FA
CRC-14/DARC	14	0x805.	0x0	true	true	0x0	0x82D
CRC-15	15	0x4599	0x0	false	false	0x0	0x59E
CRC-15/MPT1327	15	0x6815	0x0	false	false	0x1	0x2566
CRC-16/ARC	16	0x8005	0x0	true	true	0x0	0xBB3D

CRC-16/AUG-CCITT	16	0x1021	0x1D0F	false	false	0x0	0xE5CC
CRC-16/BUYPASS	16	0x8005	0x0	false	false	0x0	0xFEE8
CRC-16/CCITT-FALSE	16	0x1021	0xFFFF	false	false	0x0	0x29B1
CRC-16/CDMA2000	16	0xC867	0xFFFF	false	false	0x0	0x4C06
CRC-16/DDS-110	16	0x8005	0x800D	false	false	0x0	0x9ECF
CRC-16/DECT-R	16	0x589	0x0	false	false	0x1	0x7E
CRC-16/DECT-X	16	0x589	0x0	false	false	0x0	0x7F
CRC-16/DNP	16	0x3D65	0x0	true	true	0xFFFF	0xEA82
CRC-16/EN-13757	16	0x3D65	0x0	false	false	0xFFFF	0xC2B7
CRC-16/GENIBUS	16	0x1021	0xFFFF	false	false	0xFFFF	0xD64E
CRC-16/MAXIM	16	0x8005	0x0	true	true	0xFFFF	0x44C2
CRC-16/MCRF4XX	16	0x1021	0xFFFF	true	true	0x0	0x6F91
CRC-16/RIELLO	16	0x1021	0xB2AA	true	true	0x0	0x63D0
CRC-16/T10-DIF	16	0x8BB7	0x0	false	false	0x0	0xD0DB
CRC-16/TELEDISK	16	0xA097	0x0	false	false	0x0	0xFB3
CRC-16/TMS37157	16	0x1021	0x89EC	true	true	0x0	0x26B1
CRC-16/USB	16	0x8005	0xFFFF	true	true	0xFFFF	0xB4C8
CRC-A	16	0x1021	0xC6C6	true	true	0x0	0xBF05
CRC-16/KERMIT	16	0x1021	0x0	true	true	0x0	0x2189
CRC-16/MODBUS	16	0x8005	0xFFFF	true	true	0x0	0x4B37
CRC-16/X-25	16	0x1021	0xFFFF	true	true	0xFFFF	0x906E
CRC-16/XMODEM	16	0x1021	0x0	false	false	0x0	0x31C3
CRC-24	24	0x864CFB	0xB704CE	false	false	0x0	0x21CF02
CRC-24/FLEXRAY-A	24	0x5D6DCB	0xFEDCBA	false	false	0x0	0x7979BD
CRC-24/FLEXRAY-B	24	0x5D6DCB	0xABCDEF	false	false	0x0	0x1F23B8
CRC-31/PHILIPS	31	0x4C11DB7	0x7FFFFFFF	false	false	0x7FFFFFFF	0xCE9E46C
CRC-32/zlib	32	0x4C11DB7	0xFFFFFFFF	true	true	0xFFFFFFFF	0xCBF43926
CRC-32/BZIP2	32	0x4C11DB7	0xFFFFFFFF	false	false	0xFFFFFFFF	0xFC891918
CRC-32C	32	0x1EDC6F41	0xFFFFFFFF	true	true	0xFFFFFFFF	0xE3069283
CRC-32D	32	0xA833982B	0xFFFFFFFF	true	true	0xFFFFFFFF	0x87315576
CRC-32/MPEG-2	32	0x4C11DB7	0xFFFFFFFF	false	false	0x0	0x376E6E7
CRC-32/POSIX	32	0x4C11DB7	0x0	false	false	0xFFFFFFFF	0x765E7680
CRC-32Q	32	0x814141AB	0x0	false	false	0x0	0x3010BF7F
CRC-32/JAMCRC	32	0x4C11DB7	0xFFFFFFFF	true	true	0x0	0x340BC6D9
CRC-32/XFER	32	0xAF	0x0	false	false	0x0	0xBD0BE338

CRC називають n-бітним CRC, якщо його контрольне значення дорівнює n-бітам. Для заданого n можливі декілька CRC, кожен з різним поліномом. Такий багаточлен має найвищу степінь n, його довжина n + 1. Залишок має довжину n. Кожен алгоритм CRC має ім'я у формі CRC-n-XXX. Дізайн поліному CRC залежить від максимальної спільної довжини блоку, який має бути захищений (дані + біти CRC), бажаних функцій захисту від похибок та типу ресурсів для реалізації CRC, а також бажаної продуктивності. Часто вважають, що «кращі» поліноми CRC одержуються або з непривідних поліномів, або з непривідних поліномів, помножених на 1 + x, що додає до коду можливість виявляти всі похибки, що впливають на непарну кількість бітів. Насправді вказані фактори мають входити у вибір поліному й можуть привести до привідного поліному. Але вибір привідного поліному приведе відповідно до деякої частини пропущених похибок з-за того, що фактор-кільце має дільники нуля.

Перевага вибору примітивного поліному в якості генератору для коду CRC складається в тому, що результуючий код має максимальну спільну довжину блоку у тому смислі, що всі 1-бітні похибки в межах цієї довжини блоку мають різні залишки (сіндроми), і, тому, оскільки залишок є лінейною функциею блоку, код може виявляти всі 2-бітні похибки в межах цієї довжини блоку [5].

Часто практичне використання CRC як кода виявлення помилок ускладнюється, т.я. розробник уводить деякі спеціальні ускладнення алгоритму. Так, іноді додається фіксований бітовий шаблон до перевіряємого бітового потоку (корисно, якщо похибки синхронізації можуть вставляти 0-біти перед повідомленням, зміна, яка інакше залишила б значення перевірки незмінним). Іноді додається n 0-бітів (n – розмір CRC) до потоку бітів, який потрібно перевірити до того, як відбудеться поліноміальне ділення. Це зручно в тому, що залишок від початкового потоку бітів з доданим контрольним значенням дорівнює нулю, а тому CRC можна перевірити просто виконавши поліноміальне ділення одержаного потоку бітів та порівнявши залишок з нулем. Завдяки асоціативним і комутативним властивостям операції виключаючого АБО (XOR) у табличних алгоритмах можливе одержання результату, чисельно еквівалентного додаванню нулів без явного додавання нулів, за допомогою еквівалентного більш швидкого алгоритму. Тут іноді реалізація XOR включає фіксований бітовий шаблон у

залишок поліноміального ділення. Порядок бітів також може бути змінений. В деяких схемах молодший біт кожного байту розглядається як «перший», що під час поліноміального ділення означає «крайній зліва». Це підходить, коли передача через послідовний порт перевіряється апаратно за допомогою CRC, оскільки деякі згоди про передачу через послідовний порт спочатку передають байти молодшого розряду. Тому при використанні багатобайтових CRC може виникнути плутанина по відношенню того, чи є байт, переданий першим (або збережений у байті з найменшою адресою пам'яті), молодшим (LSB), старшим (MSB) байтом.

Лекція 7. Алгоритми стиснення інформації без втрат. Канонічний алгоритм Хафмана. Алгоритм арифметичного стиснення

Спершу розглянемо декілька *алгоритмів стиснення без втрат.* На відміну від них, алгоритми стиснення з втратами – це спочатку виділення частини інформації, що зберігається, за допомогою моделі, яка залежить від мети стиснення та особливостей джерела й приймача інформації, а потім саме стиснення без втрат. При вимірюванні фізичних параметрів таких, як яскравість, частота, амплітуда та ін., неточності неминучі, тому заокруглення припустиме. Крім того, припустимість стиснення зображення та звуку із значними втратами обумовлена особливостями сприйняття такої інформації людиною. Якщо ж передбачається комп'ютерна обробка зображення або звуку, то вимоги до втрат досить жорсткі.

Ефективність стиснення враховує не тільки **ступінь стиснення** (тобто відношення довжини нестиснених даних до довжини відповідних їм стиснених даних), а й швидкості стиснення та розтиснення. Часто користуються оберненою до степені стиснення величиною – **коефіцієнтом стиснення** (відношення довжини стиснених даних до довжини відповідних не стиснених даних).

Важливими характеристиками кожного алгоритму стиснення також є обсяги пам'яті, що необхідні для стиснення та для розтиснення (для збереження даних, що створюються та/або використовуються алгоритмом).

Розглянемо **канонічний алгоритм Хафмана** [6], який є алгоритмом без втрат. Даний алгоритм використовує тільки частоту появи однакових байтів у вхідному блоці даних. Ставить у відповідність символам вхідного потоку, що зустрічаються частіше, ланцюг бітів меншої довжини, а тим, що зустрічаються рідко – ланцюг більшої довжини (рис. 7.1). Для збору статистики базовий алгоритм вимагає двох проходів по вхідному блоку (канонічний алгоритм, але є й однопрохідні адаптивні варіанти алгоритму). Канонічний алгоритм Хафмана вимагає розташування у файлі із стисненими даними таблиці відповідності кодованих символів та кодованих ланцюгів.

З причини необхідності додавання до файлу таблиці відповідності та необхідності виконання двох проходів у базовому алгоритмі на практиці використовуються його різновиди. В них можуть або користуватися постійною таблицею, або будувати її адаптивно (тобто форму

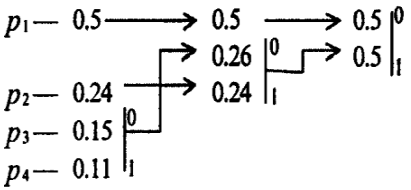


Рис. 7.1. Побудова кодів символів за їхніми частотностями p_i, $i = \overline{1,4}$ [6].

вати в процесі архівації/розархівації). Це позбавляє від необхідності двох проходів по вхідному блоку та зберігання таблиці разом з файлом. Алгоритм застосовують переважно як складову деякого іншого алгоритму стиснення. Наприклад, кодування з фіксованою таблицею застосовується в якості останнього етапу архівації в JPEG.

Основні характеристики канонічного алгоритму Хафмана [6]. Ступені стиснення: 8,1.5,1 (найкраща, середня, найгірша ступені). Симетричність за часом: 2:1 (за рахунок двох проходів по масиву стискуваних даних). Є одним з небагатьох алгоритмів, які не збільшують розміру початкових даних у найгіршому випадку, якщо не враховувати виділення простору для необхідного збереження таблиці відповідності (перекодування) разом із файлом даних.

Розглянемо наступний алгоритм стаснення без втрат – **арифметичне стиснення** [6, 7]. Він також має модифікації. Алгоритм стає більш поширеним внаслідок закінчення термінів його патентів.

Алгоритм Хафмана наближає відносні частоти появи символів у потоці частотами, що кратні степені двійки (наприклад, для символів a, b, c, d із ймовірностями 1/2, 1/4, 1/8, 1/8 будуть використані коди 0, 10, 110, 111). Арифметичне стиснення дасть кращу степінь наближення частоти. За теоремою Шенона найкраще стиснення у двійковій арифметиці отримуємо, якщо будемо кодувати символ з відносною частотою f за допомогою $-\log_2(f)$ біт. У ситуації, коли відносні частоти не є степенями двійки, стиснення стає менш ефективним (витрачаємо більше бітів, ніж це потрібно). Наприклад, якщо маємо два символи a та b з ймовірностями 253/256 та 3/256, то в ідеалі повинні витратити на ланцюг з 256 байт $-\log_2(253/256)\cdot 253 - \log_2(3/256)\cdot 3 = 23.546$, тобто 24 біта. При кодуванні за алгоритмом Хафмана закодуємо a і b як 0 та 1, тому прийдеться витратити 1 253+1 3=256 бітів, тобто у 10 разів більше. Використавши алгоритм арифметичного стиснення отримаємо результат, близький до оптимального.

Класичний варіант алгоритму [7]. Арифметичне стиснення має в основі наступну ідею. Представляємо кодований текст у вигляді дробу, при цьому будуємо дріб таким чином, щоб наш текст був представлений як можна компактніше. Для прикладу візьмемо побудову такого дробу на інтервалі [0, 1) (інтервал [0, 1) вибраний з причини зручності для пояснень). Розбиваємо його на підінтервали з довжинами, рівними ймовірностям появи символів у потоці (далі назвемо їх *діапазонами відповідних символів*).

Наприклад, стискаємо текст "КОВ.КОРОВА" (з 10 символів), тоді ймовірності появи кожного символу у тексті (в порядку спадання) та відповідні цим символам діапазони наступні:

Символ	Частота	Ймовірність	Діапазон
О	3	0.3	[0.0;0.3)
К	2	0.2	[0.3;0.5)
В	2	0.2	[0.5;0.7)
Р	1	0.1	[0.7;0.8)
А	1	0.1	[0.8;0.9)
.	1	0.1	[0.9;1.0)

Вважаємо, що наведена таблиця відома у компресорі та декомпресорі. Кодування заключається у зменшенні робочого інтервалу. Для першого символу в якості робочого інтервалу берем [0, 1). Розбиваємо його на діапазони у відповідності із заданими частотами символів (таблиця діапазонів). В якості наступного робочого інтервалу беремо діапазон, що відповідає поточному кодованому символу. Його довжина пропорційна ймовірності появи цього символу у потоці. Далі зчитуємо наступний символ. В якості вихідного беремо робочий інтервал, одержаний на попередньому кроці, знову розбиваємо його у відповідності із таблицею діапазонів. Довжина робочого інтервалу зменшується пропорційно ймовірності поточного символу, а точка початку зсувається вправо пропорційно початку діапазону для цього символу. Новий побудований діапазон береться в якості робочого і т. д.
Використовуючи отриману таблицю діапазонів кодуємо повідомлення з 10 символів "КОВ.КОРОВА":
Початковий робочий інтервал [0, 1). Для перших чотирьох символів получимо наступне.

Символ "К" [0.3; 0.5) – маємо [0.3000; 0.5000).
Символ "О" [0.0; 0.3) – маємо [0.3000; 0.3600).
Символ "В" [0.5; 0.7) – маємо [0.3300; 0.3420).
Символ"." [0.9; 1.0) – маємо [0,3408; 0.3420).

Отже, результуюча довжина інтервалу дорівнює добутку ймовірностей усіх символов, що зустрілися, а його початок залежить від порядку надходження символів у потоці. Якщо позначити діапазон символу «с» як [a[c]; b[c]), а інтервал для *i*-го кодованого символу потоку як [l_i, h_i), то алгоритм стиснення може бути записаний, наприклад, так [7]:

```
1   l[0] = 0; h[0] = 1; i = 0
2   while  не досягли кінця файлу даних DataFile
3         c = DataFile.ReadSymbol( ); i = i + 1
4         l_i = a[c]·(h_{i-1} – l_{i-1})
5         h_i = b[c]·(h_{i-1} – l_{i-1})
6   return l, h
```

Для послідовності символів "КОВ.", як довільне число з одержаного інтервалу можна взяти 0.341. Цього числа достатньо для відновлення

початкового ланцюга, якщо відома початкова таблиця діапазонів та довжина ланцюга.

Розглянемо роботу алгоритму відновлення ланцюга (декомпресію). Кожен наступний інтервал вкладений у попередній. Це значить: якщо є число 0.341, то першим символом у ланцюгу може бути тільки "К", оскільки тільки його діапазон включає це число. В якості інтервалу беремо діапазон "К" – [0.3; 0.5) і в ньому знаходиться діапазон [a[c]; b[c]), що включає 0.341. Перебираючи усі можливі символи за наведеною таблицею знаходимо, що тільки інтервал [0.3; 0.36), що відповідає діапазону для "О", включає число 0.341. Цей інтервал вибираємо як наступний робочий і т. д.

Повний алгоритм декомпресії можна записати так:

```
1   l[0] = 0; h[0] = 1; val = File.Code()
2   for (i = 1; i <= File. DataLength( ); i = i + 1)
3       for (для всіх cj)
4           li = li-1 + a[cj]·(hi-1 – li-1)
5           hi = li-1 + b[cj]·(hi-1 – li-1)
6           if (li <= val and val < hi) break
        DataFile.WriteSymbol(cj)
6   return DataFile
```

В наведеному алгоритмі: val — прочитане з потоку число (дріб), а c_j – записувані у вихідний потік символи, що розпаковуються. При використанні абетки з 256 символів c_j внутрішній цикл виконується доволі довго, але його можна прискорити. Оскільки $b[c_{j+1}]=a[c_j]$ (див. таблицю діапазонів), то l_i для c_{j+1} равно h_i, для c_j, а послідовність h_i для c_j строго зростає із зростанням j. Тобто кількість операцій у внутрішньому циклі можна скротити вдвічі, оскільки достатньо перевірити тільки одну границю інтервалу. Крім того, якщо маємо мало символів, то, відсортувавши їх у порядку зменшення ймовірностей, скоротимо число ітерацій циклу і таким чином прискоримо роботу декомпрессору. Першими будуть перевірятися символи з найбільшим значенням ймовірності. В нашому прикладі з імовірністю 1/2 будемо виходити з циклу вже на другому символі з шести. Якщо число символов велике, можливе прискорення пошуку символів іншими алгоритмами (наприклад, бінарного пошуку).

Хоч цей алгоритм досить працездібний, він буде працювати повільно у порівнянні з алгоритмом, що оперує двійковим дрібом (який задається як $0.a_1a_2\ldots a_i$ = a1\2 + a2l/4 + … + a_i1/2^i). Таким чином, при стисненні необхідно дописувати у дріб додаткові знаки доти, доки число, що одержане, не попаде до інтервалу, відповідного закодованому ланцюгу. Одержане число повністю задає закодований ланцюг за аналогічного алгоритма декодування.

Особливістю арифметичного кодування є здібність сильно стискати окремі довгі ланцюги. Наприклад, 1 біт "1" (двійкове число "0.1") для таблиці інтервалів однозначно задає ланцюг "B0000000000000..." довільної довжини (наприклад, 1000000000 символів). Тобто, якщо файл закінчується однаковими символами, наприклад, масивом нулів, то він може бути стиснений з досить вражаючим степенем стиснення. Але довжину початкового файлу тоді потрібно передавати явно декомпресору перед стисненими даними.

Наведений алгоритм може стискати тільки доволі короткі ланцюги внаслідок обмежень розрядності всіх змінних. Щоб уникнути цих обмежень, реальний алгоритм працює з цілими числами та оперує з дробом, чисельник і знаменник якого є цілими числами (наприклад, знаменник дорівнює l0000h = 65536). При цьому із втратою точності можна боротися, відслідковуючи зближення l_i і h_i та множачи чисельник і знаменник, що представляють їх дріб на деяке число (зручно на 2). С переповненням зверху можна боротися, записуючи старші біти у l_i і h_i до файлу тоді, коли вони перестають змінюватися (тобто реально вже не беруть участі у подальшому уточненні інтервалу). Перепишемо наведену раніше таблицю діапазонів з урахуванням вказаного:

Індекс (j)	Символ (c_j)	Накопичена частота	b[c_j]
0	-	-	0
1	O	3	3
2	K	2	5
3	B	2	7
4	P	1	8
5	A	1	9
6	.	1	10

Тепер запишемо алгоритм стиснення з використанням цілочисельних операцій [7]. Мінімізація втрат по точності в ньому досягається завдяки тому, що довжина цілочисельного інтервалу завжди не менше половини всього інтервалу. Коли l_i або h_i одночасно знаходяться у верхній або нижній половині (Half) інтервалу, то просто записуємо їхні однакові верхні біти до вихідного потоку, вдвічі збільшуючи інтервал. Якщо l_i та h_i наближаються до середини інтервалу, залишаючись по різні боки від його середини (для визначення цього уводять First_qtr та Third_qtr), то також вдвічі збільшуємо інтервал, записуючи біти "умовно" (означає, що реально ці біти виводяться у вихідний файл пізніше, коли стає відоме їх значення). Процедура зміни значень l_i та h_i називається *нормалізацією*, а виведення відповідних бітів - *переносом*.

Знаменник дробу у наведеному нижче алгоритмі буде дорівнювати l0000h = 65536, тобто максимальне значення h_0=65535, а так як для останнього символу b[c_{last}] є 10 (див. таблицю діапазонів), то й стала delitel для формування інтервалів теж 10. Допоміжні сталі задаються: First_qtr = 16384, Half = 32768, Third_qtr = 49152 (рядок 2). Початкове значення значення допоміжної змінної bits_to_follow для визначення кількості скидуваних бітів встановлюється 0 (рядок 3). Далі зчитуємо символ (рядок 5), знаходимо його індекс (рядок 6), формуємо діапазони (рядки 7-8) та оброблюємо варіанти переповнення (рядки 9-20).

В алгоритмі у рядках 11 та 13 відбувається виклик процедури, яку названо BitsPlusFollow (її не надаємо), для виконання переносу бітів до файлу.

```
1   l[0] = 0; h[0] = 65535; i = 0; delitel = b[c_last]·
2   First_qtr = (h[0] + 1)/4; Half = First_qtr*2; Third_qtr = First_qtr*3
3   bits_to_follow = 0
4   while не досягли кінця файлу даних DataFile
5       c = DataFile.ReadSymbol( ); i = i + 1
6       j = IndexForSymbol(c);
7       l_i = l_i-1 + b[j – 1] *(h_i-1 – l_i-1 + 1)/delitel
8       h_i = l_i-1 + b[j] *(h_i-1 – l_i-1 + 1)/delitel – 1
9       while true
10          if h_i < Half
11              BitsPlusFollow(0) // перенос бітів у файл
12          else if l_i >= Half
13              BitsPlusFollow(1) // перенос бітів у файл
```

```
14              l_i = l_i – Half; h_i = h_i – Half
15          end else
16          else  if  (l_i >= First_qtr  and  h_i  < Third_qtr)
17                bits_to_follow = bits_to_follow + 1
18                l_i = l_i – First_qtr; h_i = h_i – First_qtr
19             else break
20          end else
21      l_i = 2*l_i; h_i = 2*h_i + 1
22  return l, h
```

Для заданого рядку з 10 символів "КОВ.КОРОВА" в результаті роботи наведеного алгоритму матимемо:

Індекс (i)	Символ (c_j)	l_i	h_i	Нормалізоване l_i	Нормалізоване h_i	Результат
0	-	0	65535	-		
1	О	19660	32767	13104	65535	01
2	К	13104	28832	26208	57665	010
3	В	41937	48227	7816	58143	010101
4	.	53111	58143	15936	35967	01010111
5	К	21875	25901	21964	38071	0101011101
6	О	21964	26795	22320	41647	010101110101

Як бачимо, на символ, що має меншу ймовірність, витрачаємо в цілому більшу кількість бітів, ніж на символ з більшою.

Алгоритм декомпресії у цілочисельній арифметиці приймає наступний вигляд:

```
1   l[0] = 0; h[0] = 65535; delitel = b[c_last]·
2   First_qtr = (h[0] + 1)/4; Half = First_qtr*2; Third_qtr = First_qtr*3
3   val = CompressedFile.ReadBit( )
4   for i = 1  to  i < CompressedFile.DataLength( ) // з кроком 1
5       freq = ((val – l_i-1 + 1)*delitel – 1)/(h_i-1  – l_i-1 + 1)
6       for j = 1  to  b[j]  <= freq // пошук символу
7       l_i = l_i-1 + b[j – 1] *(h_i-1  – l_i-1 + 1)/delitel
8       h_i = l_i-1 + b[j] *(h_i-1  – l_i-1 + 1)/delitel – 1
9           while true  // обробка варіантів переповнень
10          if  h_i  < Half
11              ;  // нічого
12          else  if  l_i >= Half
```

```
13              l_i = l_i – Half; h_i = h_i – Half; val = val – Half
14          end else
15          else if (l_i >= First_qtr and h_i < Third_qtr)
16              l_i = l_i – First_qtr; h_i = h_i – First_qtr
17              val = val – First_qtr
18            else break
19          end else
20          l_i = 2*l_i; h_i = 2*h_i + 1
21          val = 2*val + CompressedFile.ReadBit( )
22      DataFile.WriteSymbol(c)
23  return DataFile
```

В алгоритмі з неточностями арифметики борються, виконуючи окремі операції над l_i и h_i синхронно у компресорі та декомпресорі. Незначні втрати точності (долі відсотку за достатньо великого файлу) і, відповідно, зменшення степені стиснення у порівнянні з ідеальним алгоритмом відбуваються під час операції ділення, при заокругленні відносних частот до цілого, при запису останніх бітів до файлу. Алгоритм можна прискорити, якщо представляти відносні частоти так, щоб дільник був степенем 2 (тобто замінити ділення операцією побітового зсуву).

Для того щоб оцінити степінь стиснення арифметичним алгоритмом конкретного рядку, потрібно знайти мінімальне число N, таке, щоб довжина робочого інтервалу при стисненні окремого символу ланцюгу була б меншою за $1/2^N$. Цей критерій означає, що в нашому інтервалі завідомо знайдеться хоч би одне число, у двійковому представленні якого після N-го знаку будуть тільки 0. Довжину ж інтервалу, порахувати просто, оскільки вона дорівнює добутку ймовірностей всіх символів.

Розглянемо приклад рядку з 2 символів a і b з ймовірностями 253/256 и 3/256. Довжина останнього робочого інтервалу для ланцюгу з 256 символів a і b з такими ймовірностями дорівнює [7]:

$$h_{256} - l_{256} = \left(\frac{253}{256}\right)^{253}\left(\frac{3}{256}\right)^{3} = \frac{253^{253}3^{3}}{2^{2048}} \approx 8.155\cdot 10^{-8}$$

Легко підрахувати, що шукане $N=24$ ($1/2^{24}= 5.96\text{-}10^{-8}$), оскільки 23 дає дуже великий інтервал (в 2 рази ширше), а 25 не є *мінімальним* числом, що задовольняє критерію. Вище було показане, що алгоритм Хафмана

кодує цей ланцюг у 256 біт. Тобто для розглянутого прикладу арифметический алгоритм дає перевагу у 10 разів над алгоритмом Хафмана і потребує менше 0.1 біту на символ.

Існує й *адаптивний алгоритм арифметичного стиснення.* Ідея такого алгоритму: перебудовувати таблицю ймовірностей b[j] у процесі стиснення та розпакування безпосередньо при одержанні чергового символу. Такий варіант алгоритму не потребує зберігання значень ймовірностей символів у вихідний файл та найчастіше дає більшу степінь стиснення. Так, наприклад, файл виду $a^{1000}b^{1000}c^{1000}d^{1000}$ (степінь означає число повторів даного символу) адаптивний алгоритм зможе стиснути ефективніше, ніж витрати 2 біти на символ. Наведений алгоритм досить просто перетворити в адаптивний: потрібно не зберігати таблицю діапазонів у файлі, а перераховувати прямо у процесі роботи компресора та декомпресора, перераховувати відносні частоти, корегуючи у відповідності з ними таблицю діапазонів. Важливо, щоб зміни у таблиці відбувалися у компресорі й декомпресорі синхронно (наприклад, після кодування ланцюгу довжини 100 таблиця діапазонів має бути точно така ж, як і після декодування ланцюгу довжини 100). Цю умову легко виконати, якщо змінювати таблицю після кодування і декодування чергового символу.

Основні характеристики арифметического алгоритма [6, 7]. Ступені стиснення: краща > 8 (можливе кодування менше біту на символ), найгірша - 1. Забезпечує кращу степінь стиснення, ніж алгоритм Хафмана (на типових даних на 1-10%). Є алгоритмом, що не збільшує розміру початкових даних у найгіршому випадку.

Лекція 8. Алгоритми стиснення без втрат. Словникові алгоритми стиснення даних

Розглянемо вхідну послідовність символів як послідовність рядків, що містять довільну кількість символів. У словникових методах рядки символів замінюють на коди, які розглядають як індекси рядків деякого словника. Рядки, що утворюють словник, називають *фразами*. При декодуванні здійснюється зворотна заміна індексу словника на відповідну фразу.

Отже, в словникових алгоритмах стиснення намагаються перетворити початкову послідовність представленням її у такій абетці, що її «літери» є фразами словника, які складаються із довільної кількості символів вхідної послідовності у довільному випадку. *Словником* вважають набір таких фраз, що мають зустрічатися в оброблюваній послідовності. Індекси фраз мають бути побудовані таким чином, щоб у середньому їх представлення займало менше місця, ніж вимагають рядки, що заміщуються, за рахунок чого і відбувається стиснення.

Зменшення розміру можливе, насамперед, внаслідок того, що у стискуваних даних зустрічається лише невелика частина усіх можливих рядків довжини n, а тому для представлення індексу фрази потрібно буде менше число бітів, ніж для представлення початкового рядку.

Якщо маємо деяку гіпотезу про частоту використання фраз або результати частотного аналізу оброблюваних даних, то можемо призначити найбільш ймовірним фразам коди меншої довжини. Крім того, якщо, наприклад, з 200 тис. різних рядків довжиною 5 мали, що половина зустрілася лише 1 раз, то такі рядки не використовуються в якості фраз при словниковому кодуванні у випадку, коли словник будується із рядків обробленої частини потоку. Спостережувані частоти частини рядків, що залишилася, швидко зменшуються із збільшенням n, що вказує на вигідність застосування статистичного кодування (коли часто використовуваним фразам ставлять у відповідність коди меншої довжини).

Звичайно просто припускають, що короткі фрази використовуються частіше довгих, й тому індекси будують, щоб довжина індексу короткої фрази була менше за довжини індексу довгої (що сприяє покращенню стиснення).

Методи Зіва-Лемпела.

Класичні алгоритми Зіва-Лемпела. Найпершими алгоритмами словникового стиснення Зіва-Лемпела (розроблені Зівом та Лемпелом) були алгоритми LZ77 (опублікований 1977 р.) та LZ78 (1978). Потім ці алгоритми вдосконалювалися, модифікувалися і в результаті виникла велика кількість нових алгоритмів та різних модифікацій.

Алгоритми LZ77 та LZ78 є універсальними алгоритмами стиснення, в яких словник формується на основі обробленої на поточний момент частини вхідного потоку (адаптивно). Різняться способом формування фраз. З цієї причини словникові алгоритми Зіва-Лемпела поділяють на

дві родини – алгоритми типу LZ77 й алгоритми типу LZ78. Іноді також зустрічаються назви методів: LZ1 і LZ2. Так як дослідження Зіва та Лемпела мали теоретичний характер і стиснення даних було лише окремим частинним результатом, то пройшов деякий час доки почали використовувати такі алгоритми, часто видозмінені. Але методи даної родини набули популярності, в їх класифікації наявна плутанина внаслідок іноді невірного віднесення конкретного алгоритму до деякої відповідної родини, а іноді небажання відкривати, який саме алгоритм отримав модифікацію (так, для комерційного програмного забезпечення поширене віднесення до модифікацій LZ77 внаслідок того, що цей алгоритм не був запатентований).

Алгоритм LZ77 [6, 7]. Алгоритм є засновником родини словникових схем – алгоритмів із *ковзаючим словником* (ковзаючим вікном). В LZ77 в якості словника використовують блок вже закодованої послідовності. Звичайно з виконанням обробки поточне положення цього блоку відносно початку послідовності постійно змінюється, словник «ковзає» по вхідному потоку даних. Ковзаюче вікно має довжину N (в ньому розміщується N символів) та складається з двох частин: послідовності довжиною W = N – *n* вже закодованих символів, яка є словником; буферу попереднього перегляду довжини *n* (звичайно *n* на порядки менше за W). Якщо на поточний момент вже закодовано t символів S_1, S_2, ...,S_t, то словником будуть W попередніх символів $S_{t-(W-1)}$, $S_{t-(W-1)+1}$, ..., S_t. В буфері знаходяться в очікуванні кодування символи S_{t+1},S_{t+2}, …, S_{t+n}. Якщо W> t, то словником буде вся оброблена частина вхідної послідовності.

Головною ідеєю алгоритму є пошук найдовшого співпадіння між рядком буфера, що починається з символу S_{t+1}, та усіма фразами словника. Ці фрази можуть починатися з будь-якого символу $S_{t-(W-1)}$, $S_{t-(W-1)+1}$, ..., S_t та виходити за межі словника, заходячи до області буферу, але мають лежати у вікні. Отже, фрази не можуть починатися з S_{t+1}, буфер не може порівнюватися сам із собою. Довжина співпадіння не повинна перевищувати розміру буферу. Одержана в результаті пошуку фраза $S_{t-(i-1)}$, $S_{t-(i-1)+1}$, ..., $S_{t-(i-1)+(j-1)}$ кодується за допомогою двох чисел: 1) **зміщення** від початку буферу, **i**; 2) довжини відповідності (**співпадіння**), **j**.

Зміщення та довжина відповідності грають роль вказівника (посилання), що єдиним чином визначає фразу. Додатково у вихідний потік записують символ s, що безпосередньо йде за рядком буферу, який

співпав. В результаті на кожному кроці кодер видає опис трьох об'єктів: зміщення та довжина відповідності (які утворюють код фрази, що дорівнює обробленому рядку буферу), а також символу s.

Далі вікно зміщується на j+1 символів вправо та здійснюється перехід до нового циклу кодування. Величина зсуву пояснюється тим, що було закодовано саме j+1 символів (j за допомогою вказівника на фразу у словнику та 1 просто за допомогою копіювання). Передача одного символу в явному вигляді дозволяє вирішити проблему обробки символів, що не зустрічалися ще жодного разу, але збільшує розмір стисненого блоку.

Наприклад, розглянемо стиснення рядку "код_ломом_колол_слова" довжиною 21 символ, де довжина буферу дорівнює 7 символам, розмір словника більше довжини рядку, що стискаємо. Нехай також: нульове зміщення зарезервували для позначення кінця кодування; символ s_t від повідає одиничному зміщенню відносно символу s_{t+1}, з якого починається буфер; якщо є декілька фраз з однаковою довжиною співпадіння, то вибирається найближча до буферу; у невизначених ситуаціях – коли довжина співпадіння нульова – зміщенню надаємо одиничне значення.

Крок	Ковзане вікно		Фраза, що співпадає	Закодовані дані		
	Словник	Буфер		i	j	s
1	-	код_лом	-	1	0	«к»
2	к	од_ломо	-	1	0	«о»
3	ко	д_ломом	-	1	0	«д»
4	код	_ломом_	-	1	0	«_»
5	код_	ломом_к	-	1	0	«л»
6	код_л	омом_ко	о	4	1	«м»
7	код_лом	ом_коло	ом	2	2	«_»
8	код_ломом	колол_с	ко	10	2	«л»
9	код_ломом_кол	ол_слов	ол	2	2	«_»
10	код_ломом_колол_	слова	-	1	0	«с»
11	код_ломом_колол_с	лова	ло	5	2	«в»
12	код_ломом_колол_слов	а	-	1	0	«а»

Для кодування **i** достатньо 5 біт, для – **j** потрібно 3 біти, нехай, також символи вимагають 1 байт для свого представлення. Тоді усього буде витрачено 12(5+3+8) = 192 біта. Початково рядок займав 21∗8 = 168 біт,

тому LZ77 кодує даний рядок дуже витратно і не вигідно. Крім того, у наведеному випущений крок кодування кінця послідовності, який вимагав би ще як мінімум 5 біт (розмір поля **i** = 5 біт).
У загальному вказаний алгоритм кодування можемо описати так:

while не досягли кінця файлу DataFile
Викликаємо процедуру пошуку максимального співпадіння find_match, що враховує обмеження на довжину співпадіння й дасть match_pos = **i**; match_len = **j**; перший символ, що не співпав, unmatch_sym = s[t+1+**j**]

Записуємо у файл стиснутих даних CompressedFile опис знайденої фрази (довжина бітового представлення **i** – Offs_Ln, довжина представлення **j** – Len_Ln, розмір символу s – 8 бітів)

```
CompressedFile.WriteBits(match_pos, Offs_Ln)
CompressedFile.WriteBits(match_len, Len_Ln)
CompressedFile.WriteBits(unmatch_sym, 8)
for l = 0 to match_len // з кроком 1
c = DataFile.ReadSymbol( ) // зчитуємо черговий символ
DeletePhrase( ) // вилучаємо із словника найстарішу фразу
AddPhrase( ) // додаємо до словника фразу, що починається з
                першого символа буфера
Move.Window(c) // зсуваємо вікно на 1 позицію, додаємо у
                кінець буферу символ c
CompressedFile.WriteBits(0, Offs_Ln) // виводимо файл стиснутих даних
```

Кодування рядка з 21 символа показало, що спосіб формування кодів у класичному LZ77 неефективний та дозволяє стискати лише порівняно довгі послідовності. Дещо покращити стиснення невеликих файлів можна використовуючи *коди змінної довжини для зміщення* **i**. Дійсно, якщо навіть використовуємо словник у 32 Кб, але закодували ще тільки 3 Кб, то зміщення реально потребує не 15, а 12 біт. Крім того, маємо суттєвий програш і внаслідок використання кодів однакової довжини для вказівника довжин співпадіння **j**. Розробниками алгоритму було показано, що LZ77 може стискати дані не гірше за будь-який спеціально для них побудований напівадаптивний словниковий метод, а подібне до одержаного вище не трапиться для послідовностей достатньо великого розміру.
Декодування стиснених даних здійснюється заміною коду на блок символів, що складається з фрази словника та явно переданого символу.

Декодер не має виконувати такі ж дії, як кодер, по зміні вікна. Фраза словника елементарно визначається за зміщенням та довжиною, тому важливою властивістю всіх алгоритмів із ковзаючим вікном є швидка робота декодеру. Звідси маємо, що для всіх алгоритмів із ковзаючим вікном характерна несиметричність за часом кодування та розкодування (при стисненні багато часу витрачаємо на пошук фраз).
Алгоритм декодування може бути, наприклад, таким:

```
while не досягли кінця файлу CompressedFile
    match_pos = CompressedFile.ReadBits(Offs_Ln) // зчитуємо зміщення i
    if  (виявлена ознака кінця файлу)
        break
    match_len = CompressedFile.ReadBits(Len_Ln) // зчитуємо довжину
                співпадіння j
     for  l = 0  to  match_len // з кроком 1 знаходимо у словнику черговий
                                символ, що співпав
          c = Dict(match_pos + l) // зчитуємо черговий символ фрази, що
                                співпала
          MoveDict(c) // зсуваємо словник на 1 позицію, додаємо у його
                                початок символ c
          DataFile.WriteSymbol(c) // виводимо у вихідний файл
c = CompressedFile.ReadBits(8) // зчитуємо символ, що не співпав
MoveDict(c) // додаємо у його початок словника
DataFile.WriteSymbol(c) // виводимо у вихідний файл
```

Розглянемо модифікацію алгоритму LZ77 – ***алгоритм LZSS*** (Сторер та Жиманські, 1982) [6, 7]. Алгоритм LZSS дозволяє досить гнучко сполучати у вихідній послідовності символи і вказівники (коди фраз), що дещо зменшує витратність за рахунок передачі одного символу (властиву LZ77). В алгоритмі відбувається додавання до кожного вказівника та символу 1-бітового префіксу f, який дозволяє розрізняти ці об'єкти (він як прапорець вказує тип та, відповідно, довжину наступних даних). В результаті можна записувати символи в явному вигляді, коли відповідний їм код має велику довжину (тому тут словникове кодування є шкідливим), а також обробляти символи, що жодного разу ще не зустрічалися.
Наприклад, закодуємо знову рядок "код_ломом_колол_слова" та порівняємо отриманий коефіцієнт стиснення для LZ77 та LZSS. Нехай пере-

писуємо символ в явному вигляді, якщо поточна довжина максимального співпадіння буферу та якої-небудь фрази словника менша або дорівнює 1. Якщо записуємо символ, то перед ним стає прапорець f із значенням 0, якщо вказівник – то ставимо 1. Якщо ж є декілька співпадаючих фраз однакової довжини, то вибираємо найближчу до буферу.

Крок	Ковзане вікно		Фраза, що співпадає	Закодовані дані			
	Словник	Буфер		f	i	j	s
1	-	код_лом	-	0	-	-	«к»
2	к	од_ломо	-	0	-	-	«о»
3	ко	д_ломом	-	0	-	-	«д»
4	код	_ломом_	-	0	-	-	«_»
5	код_	ломом_к	-	0	-	-	«л»
6	код_л	омом_ко	о	1	4	1	-
7	код_ло	мом_кол	-	0	-	-	«м»
8	код_лом	ом_коло	ом	1	2	2	-
9	код_ломом	_колол_	_	0	-	-	«_»
10	код_ломом_	колол_с	ко	1	10	2	-
11	код_ломом_ко	лол_сло	ло	1	8	2	-
12	код_ломом_коло	л_слова	л	0	-	-	«л»
13	код_ломом_колол	_слова	_	0	-	-	«_»
14	код_ломом_колол_	слова	-	0	-	-	«с»
15	код_ломом_колол_с	лова	ло	1	5	2	-
16	код_ломом_колол_сло	ва	-	0	-	-	«в»
17	код_ломом_колол_слов	а	-	0	-	-	«а»

Якщо в попередньому алгоритмі потрібно було 12 кроків, зараз для цієї ж фрази маємо 17. Причому 13 разів символи були передані явно, 4 рази використовували вказівники. За умови використання таких же довжин для **i**, **j** (**i** – 5 біт, **j** – 3 біти, символи – 1 байт для свого представлення) розмір закодованих по LZSS даних буде 13(1+8) + 4(1+5+3) = 153 біти, що є меншим за початковий розмір рядка (168 біт).

Алгоритм стиснення можна, наприклад, представити наступним чином [7], увівши константи Threshold = 2 для визначення порогу включення словникового кодування, Offs_Ln з величиною представлення зміщення 14 бітів, Len_Ln з величиною представлення довжини співпадіння у 4 біти, а змінну довжини співпадіння, як і раніше, позначити match_len.

Ковзане вікно в алгоритмі можна реалізувати за допомогою "циклічного" масиву.

```
Threshold = 2; Offs_Ln = 14; Len_Ln = 4
Win_size = (1<< Offs_Ln) // визначимо розмір вікна
Buf_size = (1<< Len_Ln) – 1 // визначимо розмір буферу
inline(Win_size, i) // виклик функції обчислення положення символу у
                        вікні
buf_sz = Buf_size // ініціалізація: заповнення буферу, пошук співпадіння
                        для 1 кроку
while не досягли кінця буферу
    if  (match_len > Buf_size)
        match_len = Buf_size
    if  (match_len <= Threshold)
        //записуємо у файл стиснених даних прапорець та символ, pos
        визначає позицію початку буферу
        CompressedFile.WriteBit(0)
        CompressedFile.WriteBits(window[pos], 8)
        match_len =1
    else  // запишемо прапорець та вказівник, що складається із зміщення
            та довжини співпадіння
        CompressedFile.WriteBit(1)
        CompressedFile.WriteBits(match_offs, Offs_Ln)
        CompressedFile.WriteBits(match_len, Len_Ln)
     for  l = 0  to  match_len-1 // з кроком 1
            DelPhraseMod(pos+ buf_sz) // вилучаємо із словника фразу, що
                                    починається з позиції Mod(pos+ buf_sz)
            if  (c = DataFile.ReadSymbol( ) == EOF
                buf_sz = buf_sz – 1 // якщо у кінці файлу, скорочуємо буфер
            else  // додамо у кінець буферу новий символ
                window[Mod(pos+ buf_sz)] = c
            pos = Mod(pos + 1)// зсуваємо вікно на 1 символ
            if  (buf_sz не пустий)
                 AddPhrase(pos, match_offs, match_len) // додамо у словник
                                    нову фразу, що починається у позиції pos
                            // разом виконується пошук максимального
                              співпадіння між буфером та фразами словника
CompressedFile.WriteBit(1)
CompressedFile.WriteBits(0, Offs_Ln) // ознака кінця файлу
```

Відповідно алгоритм декодування можна представити так:

```
while не досягли кінця буферу
    if CompressedFile.ReadBit( ) // символ, який просто виводимо у файл
                та записуємо у кінець словника (відповідає зміщенню i-1)
        c = CompressedFile.ReadBits(8)
        DataFile.WriteSymbol(c)
        window[pos] = c
        pos = Mod(pos + 1)
    else   // маємо вказівник, читаємо його
        match_ pos = CompressedFile.ReadBits(Offs_Ln)
        if  match_ pos зміження вже немає
            break  // це буде кінець файлу
        match_pos = Mod(pos – match_pos)
        match_len = CompressedFile.ReadBits(Len_Ln)
        for  l = 0  to  match_len-1 // копіюємо із словника до файлу фразу,
                                        що співпала
            c = window[Mod(match_pos+ 1)] // виводимо черговий
                                                символ, що співпав
            DataFile.WriteSymbol(c)
            window[pos] = c
            pos = Mod(pos + 1)
```

Алгоритм LZ78 [6, 7]. Являється основою цілої родини словникових методів LZ78, в якій немає ковзаного вікна та до словника розміщують не всі рядки, що зустрічаються при кодуванні, а лише перспективні з огляду на ймовірність наступного використання.

На кожному кроці у словник вставляється нова фраза, що є конкатенацією одної з фраз S словника, що має найдовше співпадіння з рядком буферу, та символу s. Символ s є наступним символом за рядком буферу, для якого знайдена фраза S, що співпадає. На відміну від родини LZ77 у словнику не може бути однакових фраз. Кодер породжує тільки послідовність кодів фраз. Кожен код складається з номеру (індексу) *n* "батьківської" фрази S, або префіксу, та символу s. На початку обробки словник пустий. Теоретично словник може зростати нескінчено, але на практиці при досягненні деякого обсягу пам’яті словник повинен очищуватися повністю або частково.

Наприклад, знову закодуємо рядок "код_ломом_колол_слова". Для LZ78 буфер зовсім не потрібний, оскільки досить легко реалізувати

пошук співпадаючої фрази максимальної довжини так, щоб послідовність незакодованих символів переглядати лише 1 раз. Але використаємо його в даному прикладі для спрощення розуміння суті. Фразу з номером 0 зарезервуємо для позначення кінця стисненого рядку, а з номером 1 – для пустої фрази словника.

Крок	Фраза, що додається у словник		Буфер	Фраза, що співпадає	Закодовані дані	
	Саме фраза	Номер фрази		S	*n*	s
1	к	2	код_лом	-	1	«к»
2	о	3	од_ломо	-	1	«о»
3	д	4	д_ломом	-	1	«д»
4	_	5	_ломом_	-	1	«_»
5	л	6	ломом_к	-	1	«л»
6	ом	7	омом_ко	о	3	«м»
7	ом_	8	ом_коло	ом	7	«_»
8	ко	9	колол_с	к	2	«о»
9	ло	10	лол_сло	л	6	«о»
10	л_	11	л_слова	л	6	«_»
11	с	12	слова	-	1	«с»
12	лов	13	лова	ло	10	«в»
13	а	14	а	-	1	«а»

Рядок закодовано за 13 кроків. На кожному кроці одержувався один код, тому стиснений рядок складається з 13 кодів. Можливе використання 15 номерів фраз (від 0 до 14), тому для представлення *n* за допомогою кодів фіксованої довжини потрібно 4 біта. Тоді розмір стиснутого рядку буде 13(4+8) = 156 бітів.

Алгоритм стиснення LZ78, в якому використано процедуру FindPhrase для пошуку у словнику фрази, що є конкатенацією батьківської фрази з номером n та символу s, можемо представити наступним чином:

```
n = 1
while не досягли кінця буферу
    s = DataFile.ReadSymbol // зчитуємо черговий символ
    FindPhrase(phrase_num, n, s) // повертає номер шуканої фрази у
       phrase_num, якщо фрази немає, то phrase_num приймає значення 1
  if  phrase_num не 1// якщо фраза є у словнику, йде далі пошук фрази
                        максимальної довжини, що співпадає
```

```
        n = phrase_num
    else  якщо фрази немає // пишемо у вихідний файл код, константа
            Index_Ln визначає довжину представлення номера n у бітах
        CompressedFile.WriteBits(n, Index_Ln)
        CompressedFile.WriteBits(s, 8)
        AddPhrase(n, s) // додамо у словник нову фразу
        n = 1  // підготовка до наступного кроку
CompressedFile.WriteBits(0, Index_Ln) // ознака кінця файлу
```

При декодуванні забезпечуємо той же порядок оновлення словника, як і при стисненні. Тоді алгоритм декодування можна представити:

```
while не досягли кінця буферу
    n = CompressedFile.ReadBits(Index_Ln) // зчитуємо індекс батьківської
                                          фрази
    if  якщо індексу n немає //
        break  // це буде кінець файлу
     s = CompressedFile.WriteBits(s, 8) // зчитуємо символ s, що не співпав
     GetPhrase(pos, len, n) // процедура пошуку у словнику позиції фрази з
                            індексом n та її довжини
      for  l = 0  to  len-1 // запис фрази з індексом n у вихідний файл
          DataFile.WriteSymbol(Dict[pos+l])
      DataFile.WriteSymbol(s) // запис символу s у вихідний файл
      AddPhrase(n, s) // додамо у словник нову фразу
```

Швидкість розкодування для алгоритмів родини LZ78 потенційно менше швидкості розкодування для алгоритмів з ковзаним вікном (за рахунок мінімальних витрат із підтримки словника у правильному стані) [7]. Але для LZ78 та його нащадків (LZW та ін.) існують ефективні практичні реалізації процедур пошуку та додавання фраз у словник, що надають переваги над алгоритмами родини LZ77 у швидкості стиснення. При реалізації алгоритму кодування повільніше декодування (співвідношення швидкостей звичайно 3:2).

В алгоритмі LZ78 виділяють наступну властивість: якщо початкові дані породжені джерелом, для якого характерна стаціонарність (якщо багатовимірні розподіли ймовірностей генерації послідовностей з n символів, де n довільне скінчене число, не змінюються в часі) та ергодичність (коли для оцінки властивостей джерела достатньо лише одної довгої

згенерованої послідовності), то коефіцієнт стиснення повільно наближається по мірі кодування до мінімально досяжного, тобто кількість бітів, що витрачені на кодування кожного символу наближатиметься у середньому до ентропії. Але на реальних довжинах даних ця характеристика алгоритму суттєво відрізняється: коефіцієнт стиснення текстів залежно від розміру текстів звичайно 3.5 – 5 бітів на символ. Крім того, на практиці часто оброблювані дані породжені джерелом, якому притаманна нестаціонарність [6-8].

Розробниками була показана наявність такої ж властивості й для класичного алгоритму LZ77, але з меншою швидкістю наближення до ентропії джерела.

Як було вказано, існує дуже багато модифікацій алгоритмів LZ77 та LZ78. Наприклад, LZW, LZH, LZP.

LZW є модифікацією LZ78. За рахунок попереднього занесення у словник всіх символів абетки вхідної послідовності результат роботи алгоритму LZW є послідовністю індексів фраз словника, тому зникає потреба в регулярній передачі одного символу в явному вигляді й алгоритм забезпечує краще стиснення, ніж батьківський LZ78.

LZH є модифікацією LZSS, що є аналогічною до LZB (що використовує змінні довжини колів для вказівників), але для стиснення зміщень і довжин відповідності використовують коди Хафмана (багато сучасних архіваторів також застосовують кодування методом Хафмана).

LZP є модифікацією LZ77 Блумом (1995). В ньому для кожного вхідного символу рядок з попередніх L символів розглядається як контекст C довжини N. За допомогою хеш-функції у словнику знаходиться одна із співпадаючих з контекстом C фраз, нехай, це фраза C': $C' = C$. Рядок S буферу порівнюється з фразою, що безпосередньо йде за C'. Якщо довжина співпадіння $L > 0$, то видається прапорець успіху $f = 1$ та S кодується через довжину L. Так як C' знаходиться детермінованим чином, то зміщення кодувати не потрібно. Якщо $L = 0$ або C' не було знайдено, видається прапорець успіху $f = 0$ та перший символ S кодується безпосередньо. Декодер має використовувати той же механізм для пошуку C'. Такі ж дії виконуються й в алгоритмі LZP1, перевагою якого є висока швидкість. У більш складних модифікаціях **LZP** процес пошуку C повторюється для контексту довжиною N-1, N-2 і т. д. У

LZP1 літерали просто копіюються, а для кодування прапорців та довжин використовують коди змінної довжини. В LZP3 застосовують досить складну схему кодування потоків прапорців, довжин та літералів на основі алгоритму Хафмана. Алгоритми LZP забезпечують високий степінь стиснення й непогані швидкості стиснення та розтиснення.

Основні характеристики родини LZ77 [6, 7]. Ступені стиснення: визначаються даними, звичайно 2-4. Алгоритми універсальні, проте найкраще підходять для стиснення різнорідних даних.

Симетричність по швидкості: біля 10:1; якщо алгоритм забезпечує добре стиснення, то декодер звичайно швидше кодера. Звичайно повільно стискають високозбиткові дані; внаслідок високої швидкості розтиснення ідеально підходять для створення дистрибутивів програмного забезпечення.

Основні характеристики родини LZ78 [6, 7]. Ступені стиснення: визначаються даними, звичайно 2-3. Алгоритми універсальні, але найкраще підходять для стиснення текстів та подібних однородних даних; погано стискають різнорідні дані.

Симетричність по швидкості: приблизно 3:2, декодер звичайно в 1,5 рази швидше кодера. Внаслідок доволі невеликої степені стиснення та невисокої швидкості декодування поступаються поширеності алгоритмів родини LZ77.

Лекція 9. Шляхи покращення стиснення для алгоритмів родин LZ

Існує два шляхи покращення стиснення алгоритмами родин Зіва-Лемпела. Це [6, 7]:

- зменшення кількості вказівників за незмінної або більшої загальної довжини закодованих фраз за рахунок більш ефективного розбиття вхідної послідовності на фрази словника;
- збільшення ефективності кодування індексів фраз словника та літералів, тобто зменшення кількості бітів, що в середньому потрібні для кодування індексу або літералу.

За першого варіанту для одного й того ж словника є велика кількість можливостей побудови набору фраз для заміщеня ними послідовності, що стискається. Використостовуючи «жадібний» розбір, за якого на кожному кроці кодер вибирає найдовшу фразу, що характерне для розбиття даних класичними алгоритмами LZ, забезпечуємо найбільшу

швидкість (якщо пошук ведеться у всьому словнику) та майже завжди найгірше стиснення. Стратегії оптимального розбору дозволяють значно покращити стиснення (10% та більше), але й суттєво уповільнюють роботу компресору.

За другого варіанту, де прагнуть до большої компактності представлення індексів словника, досягають мети за рахунок застосування більш складних алгоритмів стиснення (наприклад, контекстних методів моделювання) та мінімізації об'єму словника шляхом вилучення надлишкових співпадаючих фраз. Способом збільшення ефективності кодування літералів може бути явне статистичне моделювання ймовірностей появи літералів у сполученні з арифметичним кодуванням, яке саме й забезпечує стиснення. При цьому, як показано на дослідах [7], для покращення стиснення доцільно використовувати контекстне моделювання.

Вказані стратегії реалізовані у практичних використаннях. Архіватори 7-Zip, ACE, ARJ, ARJZ, CABARC, PKZIP, RAR, WinZip, Zip відносять до самих ефективних архіваторів, що застосовують методи Зіва-Лемпела.

Стратегія розбору LFF є однією з можливих технік покращення розбору вхідної послідовності на фрази словника LZ77 (метод кодування самого довгого рядку першим – Longest Fragment First) [7]. Сутність LFF полягає в тому, що розглядається декілька варіантів розбиття буферу на фрази словника й першим заміщується рядок, з яким співпала фраза максимальної довжини, причому рядок може починатися в будь-якій позиції буферу.

Наприклад [7], у буфері знаходиться рядок "абракадабра", а у словнику для кожної позиції буферу можна знайти наступні фрази максимальної довжини, що співпадають:

№ позиції	Фраза максимальної довжини, що співпадає	Довжина фрази
1	аб	2
2	брак	4
3	рак	3
4	ака	3
5	кадаб	5
6	адаб	4
7	даб	3

Пошук перерваний на позиції 7, оскільки вже жодне співпадіння не може бути довшим за максимальне, що зустрілося: 5. Так як спосіб кодування самого довгого рядку "кадаб" визначений, то далі для підрядку "абра", що залишився, знову шукають найдовшу фразу, що співпадаєЦе буде "бра". Рядок "а", що залишився зліва, може бути закодований як літерал. Тоді розбір буферу буде мати вигляд:

"абракадаб..." -> <а> <бра> <кадаб>.

Така послідовність з літералу та двох фраз кодується, вікно зміщується на 9 символів, а буфер одержує вигляд "ра...".

У випадку жадібного розбору буфер мав би наступне розбиття:

"абракадаб..." -> <аб> <рак> <адаб>.

Метод LFF дозволяє покращити стиснення на 0.5-1% у порівнянні із жадібним розбором [7].

Оптимальний розбір для методів LZ77. Евристичні техніки підвищення ефективності розбору вхідної послідовності (наприклад, метод LFF) не вирішують проблеми одержання дуже доброго розбиття в загальному випадку. Оптимальна стратегія розбору, що завжди забезпечує мінімізацію довжини закодованої послідовності, породжена компресорами з алгоритмом словникового стиснення родини LZ77 або LZ78 розглядалася в наукових дослідженнях [7, 8]. Для алгоритмів родини LZ77 було показано, що така задача є NP-повною. Тому пропонувалися алгоритми, що дозволяли одержати майже оптимальні рішення за деяких витрат часу. Розглянемо один з таких алгоритмів.

Алгоритм забезпечує отримання майже оптимального рішення для методів LZ77 (схожа схема використовується на практиці, наприклад, у компресорі CABARC корпорації Microsoft). У загальному випадку для кожної позиції t знаходимо всі фрази, що співпадать, довжиною від 2 до максимальної max_len. На основі інформації про коди, використовувані для стиснення фраз та літералів, можна оцінити потрібну кількість бітів для кодування кожної фрази (літералу) та ціну кодування. Це дає можливість знайти наближене до оптимального рішення за один прохід наступним чином. До масиву з іменем offsets записуватимемо посилання на фрази, що підходять для заміщення рядку буферу, довжиною від 2 до max_len (max_len є довжиною максимального співпадіння для поточної позиції t). В полі offs[len] зберігаємо зміщення фрази з довжиною 1en. Якщо є декілька варіантів фраз із одною і тою ж довжиною len, обираємо фразу з найменшою ціною price:

offsets[t].offs[len] = offset_min_price(t, len).

Розмір блоку, що обробляється – MAX_T. Для забезпечення ефективності розбору MAX_T має бути доволі великим – декілька сот байт та більше. У комірках path[t] масиву path зберігатимемо інформацію про те, як добратися до позиції t, "сплативши" мінімальну ціну (price – цілочисельна ціна найдешевшого шляху; prev_t – позиція, з якої попадаємо у t; offs – зміщення фрази, що кодується при попаданні до t, у словнику). Враховуючи вказане, можна отримати загальний вигляд алгоритму:

```
path[0].price = 0 // розпочинаємо цикл розбору
for t = 1 to t < MAX_T // з кроком 1
    path[t].price = ∞ // встановлення недосяжно великої ціни для всіх t
for t = 1 to t < MAX_T
    find_matches(offsets[t]) // пошук усіх фраз, що співпадають, для
                             рядку, що починається з позиції t
    for len = 1 to len <= offsets[t]. max_len // визначення ціни
                             переходу на визначену кількість вперед
        if len == 1
            price = get_literal_price(t)//пошук ціни кодування літералу
        else
            price = get_match_price(len, offsets[t].offs[len]) // пошук ціни
                                    кодування фрази довжиною len
        new_price = path[t].price + price // ціна шляху до t + len
        if new_price < path[t + len].price //цей шлях до t + len вигідніше
                                    за шлях, збережений у path[t + len]
            path[t + len].price = new_price
            path[t + len].prev_t = t
            if len > 1
                path[t + len].offs = offsets[t].offs[len]
```

В результаті роботи алгоритму отримуємо майже оптимальне рішення, записане у вигляді однозв'язного списку. Якщо припустити, що довжина max_len обмежується у функції findmatches так, щоб не "перескочили" позиції MAX_T, то головою списку є елемент path[MAX_T]. Шлях буде записаний у зворотному порядку, тобто фраза із зміщенням path[MAX_T].offs і довжиною MAX_T-path[MAX_T].prev_t (або літе-

рал у позиції MAX_T-1) має кодуватися останньою. На практиці застосовують евристичні правила, що прискорюють пошук за рахунок зменшення числа варіантов розгалуження, які розглядаються.
Наведений алгоритм дозволяє покращити стиснення для алгоритмів родини LZ77 на декілька відсотків [7].
Алгоритм Бендера-Вулфа (LZBW). Класичним алгоритмам родини LZ77 властива надлишковість словника. У словнику може бути, наприклад, декілька однакових фраз довжини від max_len та менше, що співпадають із рядком на початку буферу. Класичний LZ77 ніяк не використовує таку інформацію та відводить кожній фразі однаковий об'єм кодового простору, вважає їх рівноймовірними. Бендер та Вулф запропонували підхід, що дозволяє дещо компенсувати цю надлишковість алгоритмів LZ із ковзаючим вікном. В алгоритмі LZBW після знаходження фрази S, що має саме довге співпадіння з буфером, відбувається пошук самої довгої співпадаючої фрази S' серед доданих до словника пізніше S та таких, що повністю знаходяться у словнику (не заходять у область буферу). Довжина S передається декодеру як різниця між довжиною S та довжиною S'. Наприклад, якщо S = "абсд" та S' = "аб", то довжина S передається за допомогою різницевої довжини 2.
Розглянемо модифікацію алгоритму LZSS (Сторера та Жиманськи) за допомогою техніки Бендера-Вулфа на прикладі процесу кодування рядку "колол_код_ломом_колесо" починаючи з першої появи символу "м". На відміну від LZSS словникове кодування буде застосовуватися при довжині співпадіння 1 та більше.

Крок	Ковзане вікно		Фраза, що співпадає	Закодовані дані			
	Словник	Буфер		f	**i**	**j**	s
k	колол_код_ло	мом_кол	-	0	-	-	м
k+1	колол_код_лом	ом_коле	ом	1	2	2	-
k+2	колол_код_ломом	_колесо	_	1	6	1	-
k+3	колол_код_ломом_	колесо	кол	1	16	1	-
k+4	…л_код_ломом_кол	есо	-	0	-	-	е

Якщо за алгоритмом LZSS на кроці k зустріли символ "м", що відсутній у словнику, то передаємо його в явному вигляді. Зсуваємо вікно на 1 позицію. На кроці k +1 фраза, що співпадає, є "ом", а в словнику немає жодних інших фраз, що додані пізніше "ом". Тому покладемо довжину

S’ = 0. Тоді різниця j, що передається, **j** = 2. Зсуваємо вікно на 2 символи. На кроці k +2 фраза, що співпадає, є один символ "_", він останній раз зустрічався 6 символів тому. Знову різницева довжина співпадаючої фрази дорівнює її довжині, тобто одиниці. Зсуваємо вікно на 1 символ. На кроці k +3 співпадаюча фраза максимальної довжини "кол", але серед фраз, доданих до словника після "кол", є фраза "ко", тому довжина "кол" кодується як різниця 3 та 2. Якби в частині словника, що обмежена фразою "кол" зліва та початком буферу справа, не знашлося б фрази "ко" з довжиною співпадіння 2, а була б, наприклад, тільки фраза "к" з довжиною співпадіння 1, то довжина "кол" була б представлена як 3 – 1 = 2. Отже, в цій модифікації зменшується кількість бітів для представлення j, кількість кроків не змінюється.

При декодуванні необхідно підтримувати словник у тому ж вигляді, що й при кодуванні. Після одержання зміщення match_pos декодер порівнює фразу, що починається із знайденої позиції

t – (match_pos – 1) (t +1 – позиція початку буферу),

з усіма більш "новими" фразами словника, тобто тими, що знаходяться в області t – (match_pos – l), …, t. Довжина фрази відновлюється як сума максимального співпадіння та кодера різницевої довжини j. Наприклад, процедура декодування для кроку k + 3 буде виглядати так. Декодер читає зміщення **i** = 16, різницеву довжину **j** = 1 та починає порівнювати фразу "колол_...", початок якої визначається даним зміщенням **i**, з усіма фразами словника, початок яких має зміщення від 15 до 1. Максимальне співпадіння має "ко", що розташована за зміщенням 10. Тому довжина закодованної фрази є 2 + **j** = 2 + 1 = 3, тобто кодер передав вказівник на фразу "кол".

Використання техніки Бендера-Вулфа дозволяє покращити стиснення приблизно на 1%, але декодування дуже сильно уповільнюється, оскільки маємо виконувати такий же додатковий пошук для визначення довжини, що й при стисненні.

Контекстно-залежні словники [6, 7]**.**

Об’єм словника й середня довжина закодованого індекса можуть бути зменшені за рахунок використання прийомів контекстного моделювання. Використання контекстно-залежних словників покращує стиснення для алгоритмів родини LZ77 а, особливо, родини LZ78 [7].

Нехай тільки но закодували послідовність символів S невеликої довжини L, тоді на поточному кроці в якості словника використовуються

тільки рядки, що зустрілися у вже обробленій частині потоку безпосередньо після рядків, рівних S. Ці рядки утворюють **контекстно-залежний словник порядку L для S.** Якщо в словнику порядку L співпадіння виявити не вдалося, то відбувається перехід до словника порядку L-1. В кінці кінців, якщо не було знайдено фраз, що співпадають, у жодному з доступних словників, то поточний символ передається в явному вигляді.

Наприклад, при використанні техніки контекстно-залежних словників порядку L у сполученні з LZ77 після обробки послідовності "абракадабра" маємо такий склад словників порядків 2, 1 та 0 для поточного контексту:

Порядок L	Склад контекстно-залежного словника порядку L (фрази словника можуть починатися тільки у першій позиції наведених послідовностей)
2 (контекст «ра»)	кадабра
1 (контекст «а»)	бракадабра кадабра дабра бра
0 (пустий контекст)	абракадабра бракадабра ракадабра акадабра кадабра адабра дабра абра бра ра а (звичайний словник LZ77)

Доведено [7], що найкращі результати досягаються за L = 1 або 2 (в якості контексту достатньо брати 1 або 2 попередніх символи). Застосування контекстно-залежних словників дозволяє покращити стиснення (наприклад, LZSS на 1-2%, а LZW – приблизно на 10%), але відбуваються втрати в швидкості. Відповідні версії алгоритмів називають: LZ77-PM, LZW-PM. Недоліком техніки є необхідність підтримувати

доволі складну структуру контекстно-залежних словників не тільки при кодувані, а й при декодуванні.

Буферизація зміщень. Якщо була закодована фраза із зміщенням **i**, то зростає ймовірність того, що може знадобитися закодувати фрази з майже таким же зміщенням i ± δ, где δ – невелике число. Таке часто має місце при обробці двійкових даних, оскільки для них характерна наявність порівняно довгих послідовностей, що відрізняються лише в декількох позиціях. Зміщення звичайно представляється за допомогою двох (іноді більше) полів: *бази* (сегменту) та *поля додаткових бітів*, що уточнює значення зміщення відносно бази. Тому в потоці закодованих даних, породжуваному алгоритмами родини LZ77, коди фраз з одним і тим же значенням бази зміщення часто розташовують неподалік один від одного. Цю властивість можна використати для покращення стиснення, застосувавши *техніку буферизації баз зміщень*.

У буфері запам'ятовуються *m* останніх використаних баз зміщень, що різняться між собою. Буфер оновлюють за принципом списку LRU, де сама остання використана база має індекс 0, сама "стара" – індекс *m*-1. Якщо база B зміщення поточної фрази співпадає з одною із наявних у буфері баз B_i, то замість B кодується індекс **i** буферу. Потім B_i переміщується до початку списку LRU, тобто одержує індекс 0, а всі B_0, B_1, …, B_{i-1} зсуваються на 1 позицію в бік кінця списка. Інакше кодується сама база B, після чого вона додається у початок буферу як B_0, а всі буферизовані бази зміщаються на 1 позицію до кінця списку LRU, при цьому B_{m-1} вилучається з буферу.

Наприклад, візьмемо *m* рівним 2. Якщо база не співпадає з жодною наявною в буфері, то кодоване значення бази дорівнює абсолютному значенню плюс *m*. Нехай також вміст буферу дорівнює {0, 1}, тоді покрокове перетворення послідовності баз зміщень 15, 14, 14, 2, 3, 2,... буде виглядати так:

№ кроку	Абсолютне значення бази	Вміст буферу на початку кроку		Кодоване значення бази
		B_0	B_1	
1	15	0	1	17
2	14	15	0	16
3	14	14	15	0
4	2	14	15	4
5	3	2	14	5

6	2	3	2	1
7	?	2	3	?

Стан буферу після кроку 3 не змінився, оскільки всі елементи буферу мають бути різними (інакше погіршиться стиснення внаслідок внесення надлишковості в опис баз зміщень, оскільки одна й та сама база може задаватися декількома числами).

Дослідно показано, що оптимальне значення *m* лежить в межах 4-8 [7]. Вартість кодування індексу буферу звичайно нижче вартості кодування бази зміщення безпосередньо. Тому варто примусово збільшувати частоту використання буферизованих зміщень за рахунок підбору фраз з "потрібним" розташуванням у словнику.

Використання вказаної техніки покращує стиснення двійкових файлів до декількох відсотків, але не підходить для обробки текстів. Буферизацію зміщень використовують майже в усіх архіваторах, що реалізують алгоритми родини LZ77 (наприклад, 7-Zip, CABARC, WinRAR).

Сумісне кодування довжин та зміщень [7]. Між величиною зміщення і довжиною співпадіння є незначна кореляція, величина якої зростає у випадку застосування буферизації зміщень. Цю властивість можна використати, поєднавши в один метасимвол довжину співпадіння match_len та базу зміщення offset_base, та, таким чином, кодувати метасимвол на основі статистики сумісної появи визначених довжини і зміщення. Як і у випадку зміщення, в метасимвол краще включати не повністю довжину, а її квантоване значення. Наприклад, у форматі LZX (для компресора CABARC) довжини співпадіння від 2 до 8 входять у склад метасимволу безпосередньо, а всі довжини match_len > 8 відображуються в одне значення. В такому випадку довжина співпадіння довизначається шляхом окремої передачі величини match_len-9. Метасимволи довжина/зміщення та літерали входять до одної абетки, тому у спрощеному вигляді алгоритм кодування може мати такий вигляд:

```
if  match_ len >= 2 // кодуємо фразу
    if  match_ len <= 8
        metasymbol = (offset_base<<3) || (match_ len – 2)
    else
        metasymbol = (offset_base<<3) || 7
```

```
    encode_symbol(metasymbol, NON_Literal) // кодування метавимволу,
                                           вказуючи, що це не літерал, для вірного
                                           відображення значення метасимволу до
                                           абетки довжин/зміщень та літералів
    if  match_ len > 8 // довизначення довжини співпадіння
        encode_length_footer(match_ len – 9)
        encode_offset_footer(….) // кодування молодших бітів зміщення
        …..
  else  // кодування літералу у поточній позиції t +1
        encode_ symbol(window[t+1], Literal)
```

Методи контекстного моделювання. Застосування методів контекстного моделювання для стиснення даних базується на парадигмі стиснення за допомогою універсального моделювання та кодування. За нею процес стиснення складається з двох частин: моделювання; кодування (під моделюванням розуміють побудову моделі інформаційного джерела, що породило стискувані дані, а під кодуванням – відображення оброблюваних даних у стиснену форму представлення на основі результатів моделювання). "Кодувальник" створює вихідний потік, який є компактною формою представлення оброблюваної послідовності, на основі інформації, наданої "моделювальником".

Поняття "кодування" часто використовують у широкому розумінні для позначення всього процесу стиснення, тобто включаючи моделювання. Тому необхідно розрізняти поняття кодування у широкому розумінні та у вузькому (генерація потоку кодів на основі інформації моделі). Поняття "статистичне кодування" також використовують часто невизначено відносто позначення того чи іншого рівня кодування. Іноді застосовують термін "ентропійне кодування" для кодування у вузькому смислі. Позначимо процес кодування у широкому смислі "кодуванням", у вузькому – "статистичним кодуванням" ("власне кодуванням"). З теореми Шенона про кодування джерела відомо, що символ s_i, ймовірність появи якого $p(s_i)$, вигідніше всього представляти $-\log_2(p(s_i))$ бітами, при цьому середня довжина кодів може бути обчислена за формулою інтропії. Практично завжди істинна структура джерела схована, тому необхідно будувати модель джерела, яка б дозволила в кожній позиції вхідної послідовності знайти оцінку $p(s_i)$ ймовірності появи кожного символу s_i абетки вхідної послідовності. Оцінка ймовірностей символів при моделюванні відбувається на основі відомої статистики

та, можливо, апріорних припущень, тому часто кажуть про задачу статистичного моделювання. Можна сказати, що моделювальник передбачає ймовірність появи кожного символу в кожній позиції вхідного рядку, звідси й існує ще назва цієї компоненти – "предиктор". На етапі статистичного кодування виконується заміщення символу s_i з оцінкою ймовірності появи $p(s_i)$ кодом довжини $-\log_2 p(s_i)$ бітів.

Наприклад, припустимо, що стискаємо послідовність символів абетки {"0","1"}, породжену джерелом без пам'яті, ймовірності генерації символів: p("0") = 0.4, p("1") = 0.6. Нехай модель дає такі оцінки ймовірностей: q("0") = 0.35, q("1") = 0.65. Ентропія H джерела $-p('0')\lg p('0') - p('1')\lg p('1') = -0.4\lg 0.4 - 0.6\lg 0.6 \approx 0.971$ бітів, «ентропія» моделі $-q('0')\lg q('0') - q('1')\lg q('1') = -0.35\lg 0.35 - 0.65\lg 0.65 \approx 0.934$.

Здається модель забезпечує краще стиснення, ніж за формулою Шенона, але дійсні ймовірності не змінилися. Якщо виходити з ймовірностей p, то "0" маємо кодувати $-\lg 0.4 \sim 1.322$ біти, для "1" $-\lg 0.6 \sim 0.737$ біти. Для оцінок ймовірностей q маємо $-\lg 0.35 \sim 1.515$ біти та $-\lg 0.65 \sim 0.621$ біти відповідно. При кожному кодуванні на основі інформації моделі у випадку "0" будемо втрачати 1.515 – 1.322 = 0.193 біти, а у випадку "1" вигравати 0.737 – 0.621 = 0.116 біти. З урахуванням ймовірностей появи символів середній програш за кожного кодування буде: 0.4*0.193 -0.6*0.116 = 0.008 біти. Тому чим точніша оцінка ймовірностей появи символів, тим більше коди відповідають оптимальним, тим краще стиснення.

Правильність декодування забезпечується використанням точно такої ж моделі, що була застосована при кодуванні. Тому при моделюванні для стиснення даних не можна користуватися інформацією, що невідома декодеру. На практиці арифметичний кодер дозволяє сгенерувати стиснену послідовність, довжина якої часто лише на десяті долі відсотку перевищує теоретичну довжину, розраховану на основі формули Шенона. А застосування його сучасної модифікації – інтервального кодера – дозволяє здійсняти власне кодування дуже швидко. Швидкість статистичного кодування зараз складає мільйони символів на секунду на ПК. Тому підвищення точності моделей є фактично єдиним способом суттєвого покращення стиснення.

Класифікація стратегій моделювання (за способом побудови та оновлення моделі). Виділяють 4 варіанти моделювання: статичне; напівадаптивне; адаптивне (динамічне); блочно-адаптивне [7].

При *статичному* моделюванні для будь-яких оброблюваних даних використовують одну й ту ж модель (не відбувається адаптація моделі до особливостей стискуваних даних). Опис наперед побудованої моделі зберігається у структурах даних кодера й декодера, таким чином досягається однозначність кодування та відсутність необхідності в явній передачі моделі. Недоліком є: можна отримувати погане стиснення та навіть збільшити розмір представлення, якщо оброблювані дані не відповідають вибраній моделі. Тому таку стратегію використовують тільки у спеціалізованних додатках, коли тип стискуваних даних не міняється та наперед відомий.

Напівадаптивне стиснення є розвитком стратегії статичного моделювання. В ньому для стиснення заданої послідовності вибирається або будується модель на основі аналізу саме оброблюваних даних. Кодер має передавати декодеру не тільки закодовані дані, а й опис використаної моделі. Якщо модель вибирається з наперед створених та відомих кодеру та декодеру, то це просто порядковий номер моделі. А якщо модель була налаштована або побудована при кодуванні, то необхідно передавати або значення параметрів налаштувань, або модель повністю. У загальному випадку підхід дає краще стиснення, ніж статичний, так як забезпечує підлаштування до природи оброблюваних даних, зменшуючи ймовірність значної різниці між передбаченням моделі та реальною поведінкою потоку даних.

В *адаптивному* моделюванні у процесі кодування модель змінюється за заданим алгоритмом після стиснення кожного символу. Однозначність декодування досягається тим, що від початку кодер і декодер мають ідентичну й звичайно дуже просту просту модель, а модифікація моделі при стисненні та розтисненні здійснюється однаковим чином. Звичайно підхід забезпечує не гірше стиснення, ніж напівадаптивне моделювання.

Блочно-адаптивне моделювання можна розглядати як частковий випадок адаптивної стратегії. Залежно від конкретного алгоритму оновлення моделі, оцінки ймовірностей символів, методу статистичного кодування та власне даних зміна моделі після обробки кожного символу може бути спряжена з: втратою стійкості оцінок, великими обчислювальними витратами на оновлення моделі, великими витратами пам’яті для зберігання структур даних, що забезпечують швидку модифікацію

моделі. Тому оновлення моделі може виконуватися після обробки цілого блоку символів, у загальному випадку змінної довжини. Для забезпечення правильності розтиснення декодер має виконувати таку ж послідовність дій з оновлення моделі, що й кодер, або кодеру необхідно передавати разом із стисненими даними інструкції з модифікації моделі. Останнє доволі часто використовується за блочно-адаптивного моделювання для прискорення процесу декодування за рахунок коефіцієнту стиснення.
На практиці часто використовують гібридні схеми.

Лекція 10. Застосування контекстного моделювання. Алгоритми PPM

Для застосування контекстного моделювання необхідно вирішити задачу оцінки ймовірностей появи символів у кожній позиції оброблюваної послідовності. Щоб розтиснення відбулося без втрат, можна користуватися тільки тією інформацією, що в повній мірі відома кодеру та декодеру. Звичайно це означає, що оцінка ймовірності чергового символу має залежати тільки від властивостей вже обробленого блоку даних.
Найпростіший спосіб оцінки реалізується за допомогою напівадаптивного моделювання й полягає у попередньому підрахунку безумовної частоти появи символів у стискуваному блоці. Отриманий розподіл ймовірностей використовується для статистичного кодування всіх символів блоку. Якщо, наприклад, таку модель застосувати для стиснення тексту російською мовою, то у середньому на кодування кожного символу буде витрачено приблизно 4.5 біти, що є середньою довжиною кодів для моделі, яка базується на використанні безумовного розподілу літер у тексті [7]. В цьому випадку досягається степінь стиснення 1.5 по відношенню до тривіального кодування, де всім символам призначаються коди однакової довжини.
Аналіз поширених типів даних, наприклад, текстів на природніх мовах, виявляє сильну залежність ймовірності появи символів від безпосередньо їм передуючих. Інакше кажучи, більша частина даних, з якими стикаємося, породжується джерелами з пам’яттю. Нехай відомо, що стискуваний блок є текстом українською мовою. Якщо, наприклад,

рядок з трьох тільки що оброблених символів дорівнює "_ци", то поточний символ скоріше всього входить до групи: "г" ("циган"), "п" ("ципа"), "р" ("цирк"), "ц" ("циц"), "к" ("цикорій").

На практиці відомо, що у випадку посимвольного кодування при використанні інформації про 1 безпосередньо передуючий символ, наприклад, для російських текстів досягається середня довжина кодів у 3.6 біти, за урахування 2 останніх – вже порядку 3.2 біти. В першому випадку моделюються умовні розподіли ймовірностей символів, що залежать від значення рядку з одного безпосередньо передуючого символу, в другому – залежні від рядку з двох передуючих символів.

Під *контекстним моделюванням* будемо розуміти оцінку ймовірності появи символу (елементу, піксела, семпла та ін.) в залежності від безпосередньо йому передуючих або контексту.

Часто виокремлюють "лівосторонні" та "правосторонні" контексти, тобто послідовності символів, що безпосередньо знаходяться біля поточного символа зліва та справа відповідно. Розглядати будемо лівосторонній.

Якщо довжина контексту обмежена, то такий підхід називають *контекстним моделюванням обмеженого порядку*, при цьому під порядком розуміється максимальна довжина використовуваних контекстів N. Наприклад, при моделюванні порядку 3 для останнього символу "о" в послідовності "...молоко..." контекстом максимальної довжини 3 є рядок "лок". При стисненні цього символу під "поточними контекстами" могужуть розуміти "лок", "ок", "к", а також пустий рядок " ". Всі ці контексти довжини від N до 0 називають *активними контекстами* в тому смислі, що при оцінці символу може бути використана накопичена для них статистика.

Замість "контекст довжини о, о < N" звичайно кажуть "контекст порядку о". Техніка контекстного моделювання саме обмеженого порядку одержала найбільший розвиток.

Оцінки ймовірностей при контекстному моделюванні будуються на основі звичайних лічильників частот, пов’язаних з поточним контекстом. Якщо обробили рядок "абсабвбабс", то для контексту "аб" лічильник символу "с" дорівнює двом (символ "с" з’явився у контексті "аб" 2 рази), символу "в" – одиниці. На основі цієї статистики можна стверджувати, що ймовірність появи "с" після "аб" дорівнює 2/3, а

ймовірність появи "в" – 1/3, тобто оцінки формуються на основі вже проглянутої частини потоку.

У загальному випадку для кожного контексту кінцевої довжини o <N, що зустрічається в оброблюваній послідовності, створюється контекстна модель (КМ). Будь-яка КМ включає до себе лічильники всіх символів, що зустрілися у відповідному до неї контексті, тобто одразу після рядку контексту. Після кожної появи якого-небудь символу s у розглянутому контексті відбувається збільшення значення лічильника символу s у відповідній контексту КМ. Звичайно лічильники ініціалізуються нулями. На практиці лічильники звичайно створюються по мірі появи у заданому контексті нових символів (лічильників, які жодного разу не зустрілися у заданому контексті символів не існує).

Під ***порядком КМ*** розуміють довжину відповідного до неї контексту. Якщо порядок КМ дорівнює o, то позначимо таку КМ як "КМ(o)". Крім звичайних КМ, використовують контекстну модель мінус 1-го порядку КМ(-1), що присвоює однакову ймовірність всім символам абетки потоку, що стискається. Для 0-го та мінус 1-го порядку контекстна модель одна, а КМ більшого порядку може бути декілька. КМ(0) та КМ(-1) завжди активні.

Іноді кажуть про "батьківські" та "дочкові" контексти. Для контексту "к" дочковими є "ок" та "лк", оскільки вони утворені зчепленням (конкатенацією) одного символу і контексту "к". Аналогічно для контексту "лок" батьківським є контекст "ок", а контекстами-предками – "ок", "к", " ". Очевидно, що "пустий" контекст " " є предком для всіх. *Сукупність КМ утворює модель джерела даних. Під порядком моделі розуміють максимальний порядок використовуваних КМ.*

Приклад обробки рядку "абсабвбабс" висвітлює проблеми контекстного моделювання:

• як вибирати підходящий контекст (контексти) серед активних з метою одержання більш точної оцінки (поточний символ може краще передбачатися не контекстом 2-го порядку "аб", а контекстом 1-го порядку "б");

• як оцінювати ймовірність символів, що мають нульову частоту (наприклад, "г").

На практиці також потрібно враховувати, що більшість реальних даних характеризується неоднорідністю, нестабільністю сили і виду статистичних взаємозв’язків, тому "стара" статистика контекстно-залежних

частот появи символів навіть шкодить. А тому моделі, що будують оцінку тільки на основі інформації КМ максимального порядку N, забезпечують порівняно низьку точність передбачення. Крім того, зберігання моделі більшого порядку вимагає багато пам'яті. Якщо в моделі використовують для оцінки тільки КМ(N), то іноді такий підхід називають *"чистим" контекстним моделюванням порядку N*.

Реально використовувані файли часто мають порівняно невеликий розмір, тому для покращення їх стиснення необхідно враховувати оцінки ймовірностей, що отримані на основі статистики контекстів різних довжин. Техніка об'єднання оцінок ймовірностей, відповідних окремим активним контекстам, в одну оцінку називається *змішуванням*. Відомо декілька способів виконання змішування.

Розглянемо модель довільного порядку N. Якщо $q(s_i|o)$ є ймовірність, присвоєна в активній КМ(o) символу s_i абетки стискуваного потоку, то *змішана ймовірність* $q(s_i)$ обчислюється у загальному випадку як:

$$q(s_i) = \sum_{o=-1}^{N} w(o) q(s_i \mid o),$$

де $w(o)$ – вага оцінки КМ(o). Оцінка $q(s_i|o)$ звичайно визначається через частоту символу s_i: $q(s_i \mid o) = \dfrac{f(s_i \mid o)}{f(o)}$ (де $f(s_i \mid o)$ – частота появи символу s_i у відповідному контексті порядку o; $f(o)$ – загальна частота появи відповідного контексту порядку o в обробленій послідовності.

Вірніше було б писати не $f(s_i|o)$, а $f(s_i|C_{j(o)})$, тобто "частота появи символу s_i у КМ порядку o з номером j(o)", оскільки контекстних моделей порядка o може бути велика кількість. Але при стисненні кожного поточного символу розглядаємо лише одну КМ для кожного порядку, так як контекст визначається безпосередньо підходящим зліва до символу рядком визначеної довжини. Інакше кажучи, для кожного символу маємо набір з N+1 активних контекстів довжини від N до 0, кожному з яких однозначно відповідає тільки одна КМ, якщо вона взагалі є.

Якщо вага $w(-1) > 0$, то це гарантує успішність кодування будь-якого символу вхідного потоку, так як наявність КМ(-1) дозволяє завжди отримати ненульову оцінку ймовірності та код кінцевої довжини.

Розрізняють моделі з *повним змішуванням*, коли передбачення визначається статистикою КМ всіх використовуваних порядків, та з *частковим змішуванням* – коли передбачення не визначається статистикою КМ всіх використовуваних порядків.

Наприклад, розглянемо процес оцінки другого символу "л", що зустрівся у блоці "молочное_молоко". Вважаємо, що модель працює на рівні символів. Нехай використовується контекстне моделювання порядку 2 та відбувається повне змішування оцінок розподілів ймовірностей в КМ 2-го, 1-го і 0-го порядку з вагами 0.6, 0.3 і 0.1. Вважаємо, що на початку кодування у КМ(0) створюються лічильники для всіх символів абетки {"м", "о", "л", "ч", "н", "е","_", "к"} й ініціалізуються одиницею; лічильник символу після його обробки зростає на 1. Для поточного символу "л" є контексти "мо", "о" та пустий (0-го порядку). На даний момент для них накопичена статистика, показана у табл. 10.1.

Табл. 10.1. Накопичена статистика блоку "молочное_молоко" [7]

Символи		"м"	"о"	"л"	"ч"	"н"	"е"	"_"	"к"
КМ порядку 0 (контекст " ")	Частоти	3	5	2	2	2	2	2	1
	Накопичені частоти	3	8	10	12	14	16	18	19
КМ порядку 1 (контекст "о")	Частоти	-	-	1	1	-	1	-	-
	Накопичені частоти	-	-	1	2	-	3	-	-
КМ порядку 2 (контекст "мо")	Частоти	-	-	1	-	-	-	-	-
	Накопичені частоти	-	-	-	-	-	-	-	-

Тоді оцінка ймовірності для символа "л" буде дорівнювати

$$q("л") = 0.1\frac{2}{19} + 0.3\frac{1}{3} + 0.6\frac{1}{1} = 0.71.$$

У загальному випадку для однозначного кодування символу "л" таке оцінювання потрібно зробити для всіх символів абетки.

З одного боку декодер не знає, чому дорівнює поточний символ, з іншого боку оцінка ймовірності не гарантує унікальності коду, а лише

задає його довжину. Тому статистичне кодування виконується на основі накопиченої частоти (наприклад, як у арифметичному стисненні). Якщо кодувати на основі статистики тільки 0-го порядку, то існує взаємно-однозначна відповідність між накопиченими частотами з діапазону (8,10] та символом "л", що не має місця у випадку просто частоти (частоту 2 мають ще 4 символи). Подібне має місце і у випадку оцінок, що одержуються частковим змішуванням. Очевидно, що успіх застосування змішування залежить від способу вибору ваг *w*(o). Простий шлях полягає у використанні заданого набору фіксованих ваг КМ різних порядків за кожної оцінки. Альтернативним є адаптація ваг по мірі кодування. Підлаштування може заключатися у наданні все більшої значущості КМ все більших порядків або спробі вибрати найкращі ваги на основі визначених статистичних характеристик останнього обробленого блоку даних. Практичний розвиток одержали *методи неявного зважування* внаслідок їх меншої обчислювальної складності.

Техніка *неявного зважування* пов'язана з уведенням допоміжного символу відходу (escape). *Символ відходу* є квазісимволом і не має належати абетці стискуваної послідовності. Фактично він використовується для передачі декодеру вказівок кодеру. Ідея полягає в наступному: якщо використовувана КМ не дозволяє оцінити поточний символ (його лічильник дорівнює 0 в цій КМ), то на вихід посилається закодований символ відходу й робиться спроба оцінити поточний символ в іншій КМ, якій відповідає контекст іншої довжини. Звичайно спроба оцінки починається з КМ найбільшого порядку N, потім у визначеному порядку здійсняється перехід до КМ менших порядків.

Статистичне кодування символу відходу виконується на основі його ймовірності, так називаємої *ймовірності відходу*. Символи відходу породжуються не джерелом даних, а моделлю, тому їх ймовірність може залежати від характеристик стискуваних даних, властивостей КМ, з якої відбувається відхід, влістивостей КМ, на яку йде відхід і т.д. Як можна оцінити цю ймовірність, рахуючи, що кінцевий критерій якості – покращення стиснення? *Ймовірність відходу – це ймовірність появи в контексті нового символу*. Тоді фактично необхідно оцінити вирогідність появи події, що жодного разу не відбувалася. Теоретичного фундаменту для вирішення цієї проблеми поки що немає, було запропоновано декілька підходів, добре працюючих на практиці. Експериментально доведено, що моделі з неявним зважуванням стійкі

відносно використовуваного методу оцінки ймовірності відходу, тобто вибір деякого способу обчислення цієї величини не впливає на коефіцієнт стиснення кардинально.

Алгоритми PPM [7]. Техніка контекстного моделювання Prediction by Partial Matching (передбачення за частковим співпадінням), запропонована Клірі та Уіттеном, є одною з самих відомих підходів до стиснення якісних даних та самою популярною серед контекстних методів. Алгоритми, що відносяться до PPM, незмінно забезпечують у середньому найкраще стиснення при кодуванні даних різних типів та є стандартом при порівнянні універсальних алгоритмів стиснення.

Опишемо деякий узагальнений алгоритм PPM. Як і у випадку інших контекстних методів, для кожного контексту, що зустрічається в оброблюваній послідовності, створюється своя контекстна модель КМ. При цьому під контекстом розуміється послідовність елементів одного типу – символів, пікселів, чисел, але не набір різнорідних об'єктів. Замість слова "елемент" застосовуватимемо "символ". Кожна КМ включає до себе лічильники всіх символів, що зустрічалися у відповідному контексті.

PPM відносять до адаптивних методів моделювання. Наперед кодеру і декодеру поставлена у відповідність початкова модель джерела даних. Вважаємо, що вона складається з КМ(-1), шо присвоює однакову ймовірність всім символам абетки вхідної послідовності. Після обробки поточного символу кодер і декодер змінюють свої моделі однаковим чином, нарощуючи величину оцінки ймовірності символу, що розглядається. Наступний символ кодується (декодується) на основі нової, зміненої моделі, після чого модель знову модифікується і т. д. На кожному кроці забезпечується ідентичність моделі кодера і декодера за рахунок застосування однакового механізму її оновлення.

В PPM використовують неявне зважування оцінок. Спроба оцінки символу починається з КМ(N), де N є параметром алгоритму і називається порядком PPM-моделі. У випадку 0-ї частоти символу в КМ поточного порядку здійснюється перехід до КМ меншого порядку за рахунок використання стратегії відходів.

Фактично, ймовірність відходу – це сумарна ймовірність всіх символів абетки вхідного потоку, які ще жодного разу не появилися у контексті. Будь-яка КМ має давати не рівну 0 оцінку ймовірності відходу. Вик-

лючення з цього правила можливі тільки коли значення всіх лічильників КМ для всіх символів абетки не є 0, тобто будь-який символ може бути оцінений у розглянутому контексті. Оцінка ймовірності відходу традиційно є одною з основних проблем алгоритмів з неявним зважуванням.

Спосіб моделювання джерела за домогою класичних алгоритмів PPM спирається на такі припущення про природу джерела:

1) джерело є марковським з порядком N, тобто ймовірність генерації символа залежить від N попередніх символів і тільки від них;

2) джерело має особливість – чим ближче розташований один із символів контексту до поточного символу, тим більше кореляція між ними.

Механізм відходів відпочатку розглядався як допоміжний прийом, що дозволяє вирішити проблему кодування символів, які жодного разу не зустрічалися у контексті порядку N. В ідеалі, досяжному після обробки достатньо довгого блоку, жодного звернення до КМ порядку менше N відбуватися не повинно. Інакше кажучи, віднесення класичних алгоритмів PPM до методів, що виконують зважування, нехай і неявним чином, є не дуже коректним.

При стисненні чергового символу виконуються такі дії. Якщо символ s оброблюється з використанням PPM-моделі порядку N, то спершу розглядають КМ(N). Якщо вона оцінює ймовірність s числом, не рівним 0, то сама й використовується для кодування s. Інакше видається сигнал у вигляді esc-символу й на основі меншої за порядком КМ(N-1) відбувається чергова спроба оцінити ймовірність s. Кодування йде через відхід до КМ менших порядків доки s не буде оцінений. КМ(-1) гарантує, що це відбудеться. Таким чином, кожен символ кодується серією кодів esc-символу, за якою йде код самого символу. Тому ймовірність відходу теж можна розглядати як ймовірність переходу до контекстної моделі меншого порядку. Якщо в процесі оцінки виявиться, що поточний розглянутий контекст зустрічаєься вперше, для нього створюється КМ. При оцінці ймовірності символу у КМ порядку o < N можна виключити з розгляду всі символи, що містяться у КМ(o+1), оскільки жоден з них не є символом s. Для цього в поточній КМ(o) потрібно замаскувати (тимчасово встановити в нуль, значення лічильників всіх символів, що є в КМ(o+1). Таку техніку називають методом виключення.

Після власне кодування символу звичайно здійснюється оновлення статистики всіх КМ, що використані при оцінці його ймовірності, за

виключенням статичної КМ(-1). Такий підхід називають методом виключення при оновленні. Найпростішим способом модифікації є інкремент лічильників символу в цих КМ.

Приклад роботи алгоритму PPM. Нехай маємо послідовність символів "абвавабввбббв" абетки {"а", "б", "в", "г"}, лічильник esc-символу дорівнює 1 для всіх КМ, при оновленні моделі лічильники символів збільшуються на 1 у всіх активних КМ, застосовується метод виключення, максимальна довжина контексту дорівнює 3 (тобто N = 3).

Спочатку модель складається з КМ(-1), в якій лічильники всіх символів абетки мають значення 1. Стан моделі обробки послідовності з 13 символів "абвавабввбббв" представлений на рис. 10.1, де прямокутни ками позначені контекстні моделі, при цьому для кожної КМ вказаний курсивом контекст, символи, що зустрілися у контексті, їхні частоти.

Нехай поточний символ дорівнює "г" (є 14-тим символом), тоді процес його кодування буде виглядати так. Спершу розглядаємо контекст 3-го порядку "ббв". Раніше він не зустрічався, тому кодер, нічого не шле на вихід, переходить до аналізу статистики для контексту 2-го порядку. В цьому контексті ("бв") зустрічалися символ "а" й символ "в", лічильники яких у відповідній КМ дорівнюють 1, тому ecs-символ кодується з ймовірністю 1/(2+1), де у знаменнику число 2 – це частота, що спостерігалася, появи контексту "бв", 1 – значення лічильника ecs-символу. В контексті 1-го порядку "в" двічі зустрічався символ "а", який виключається (маскується), 1 раз "в", теж маскується, та 1 раз "б", тому оцінка ймовірності відходу дорівнюватиме 1/(1+1). У КМ(0) символ "г" також оцінити не можна, причому всі наявні в цій КМ символи "а", "б", "в" виключаються, так як вже зустрічалися у КМ більш високого порядку. Тому ймовірність відходу одержимо рівною 1. Цикл оцінювання завершується на рівні КМ(-1), де "г" на цей час єдиний символ, що доки не зустрічався, тому він получає ймовірність 1 та кодується за допомогою 0 бітів. Отже, за використання якісного статистичного кодувальника для представлення "г" потребуємо в цілому приблизно 2.6 бітів.

Перед обробкою наступного символу створюється КМ для рядку "ббв" і відбувається модифікація лічильників символу "г" у створеній та в усіх переглянутих КМ. В даному випадку потрібна зміна КМ всіх по рядків від 0 до N. Таблиця 10.2 демонструє оцінки ймовірностей, які мали би бути використані при кодуванні всіх символів абетки у поточній позиції.

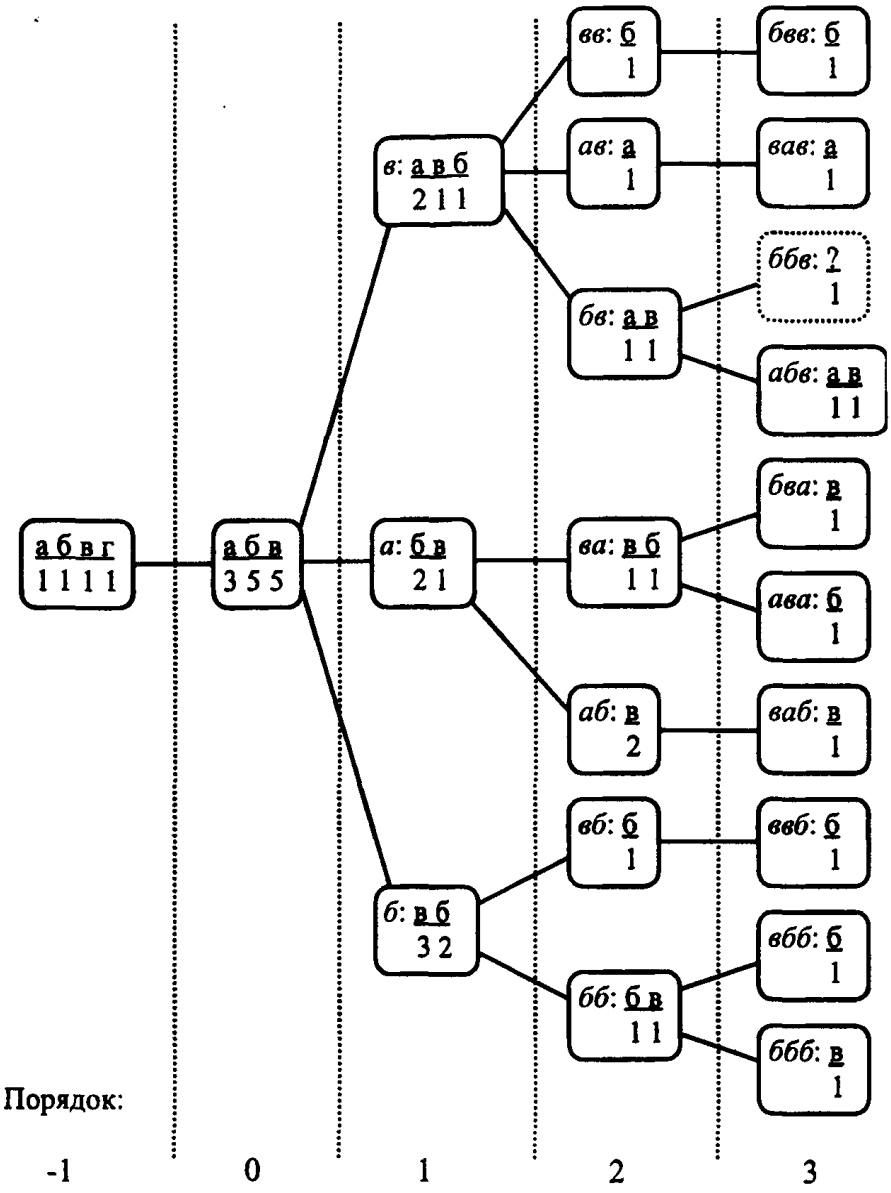


Рис. 10.1. Стан моделі при обробці послідовності "абвавабвббббв" [7].

Табл. 10.1. Оцінки ймовірностей для поточної позиції [7]

Символ s	Послідовність оцінок для КМ від 3 до -1 порядку					Загальна оцінка ймовірності q(s)	Представлення вимагає бітів
	"3	2	1	0	-1		
	"ббв"	"бв"	"в"	" "			
"а"	-	$\frac{1}{2+1}$	-	-	-	1/3	1.6
"б"	-	$\frac{1}{2+1}$	$\frac{1}{1+1}$	-	-	1/6	2.6
"в"	-	$\frac{1}{2+1}$	-	-	-	1/3	1.6
"г"	-	$\frac{1}{2+1}$	$\frac{1}{1+1}$	1	1	1/6	2.6

Алгоритм декодування є симетричним алгоритму кодування. Після декодування символу у поточній КМ перевіряємо, чи не є він esc-символом, якщо це так, то виконується перехід до КМ порядком нижче. Інакше вважають, що символ відновлений, він записується у декодований потік та відбувається перехід до наступного кроку. Зміст процедур оновлення лічильників, створення нових контекстних моделей, допоміжних дій та послідовність їх застосування мають бути строго однаковими при кодуванні й декодуванні, інакше можлива десинхронізація копій моделі кодера і декодера, що приведе до помилкового декодування якого-небудь символу, а починаючи з цієї позиції вся частина стисненої послідовності, що залишилася, буде розтиснена невірно.

Різниця між кодами символів, оцінки ймовірності яких однакові, досягається за рахунок того, що PPM-передбачальник передає кодувальнику накопичені частоти (накопичені ймовірності) оцінюваного символу та його сусідів або кодові простори символів.

Так, наприклад, для контексту "бв" з прикладу маємо:

Символ	Частота	Оцінка ймовірності	Оцінка накопиченої ймовірності	Кодовий простір
"а"	1	1/3	1/3	[0 … 0.33)
"б"	0	-	-	-
"в"	1	1/3	2/3	[0.33 … 0.66)
"г"	0	-	-	-
Відхід	1	1/3	1	[0.66 … 1)

Гарний кодувальник повинен відобразити символ s з оцінкою ймовірності q(s) у код довжини $\log_2 q(s)$, що й забезпечить стиснення всієї послідовності в цілому.
В узагальненому вигляд алгоритм кодування можна записати так:

1. Ініціалізація контексту довжини N (визначається множина активних контекстів (попередніх символів) довжини не вище N).
2. **while** (не досягнемо кінця файлу)
   c = DataFile.ReadSymbol ( ) //зчитуємо поточний символ
   order = N // поточний порядок КМ
   success = 0 // успішність оцінки у поточній КМ
   2.1. Пошук КМ для контексту поточної довжини доки не досягнемо не рівної 0 успішності
   CM = ContextModel/FindModel(context, order)
   2.2. Пошук поточного символу c в цій КМ, його накопиченої частоти CFreg (або накопиченої частоти ecs-символу)
   success = CM.EvaluateSymbol(c, CFreg, counter) // counter – посилання на лічильник символу
   2.3. Занесення у стек КМ та вказівника на лічильник для наступного оновлення моделі
   Steak.Push(CM, counter)
   2.4. Кодування символу c або ecs-символу
   StatCoder.Encode(CM, CFreg, counter)
   2.5. Зменшення порядку КМ
   order = order – 1
3. **while** (не успішна оцінка у поточній КМ)
   Оновлення моделі (КМ, лічильники і т.д.).
   Оновлення контексту (зсув вліво, додавання справа символу c)

Найкращі результати алгоритми PPM показують на текстах: високий коефіцієнт стиснення за високої швидкості (наприклад, компресори PPMd та PPMonstr). Крім того, для задачі максимізації степені стиснення визначених даних, скоріше за все PPM-подібний алгоритм буде найкращим в якості основи спеціалізованого компресора [7]. Перевага PPM: можливість одержання гарної статистичної моделі обробленої послідовності якісних даних (або джерела, що їх сгенерувало). Модель, що дозволяє ефективно передбачати невідомі символи повідомлення, можна застосовувати не тільки для стиснення, а й для вирішення задач

корегування тексту, розпізнавання мови, класифікації типу текста, семантичного аналізу текста, шифрування. Недоліки реалізації підходу PPM: повільне декодування (звичайно на 5-10% повільніше кодування); несумісність кодера й декодера при зміні алгоритму оцінки (а, наприклад, алгоритми родини LZ77 припускають суттєву модифікацію кодеру без необхідності виправлення декодера); повільна обробка малонадлишкових даних; найкраще стиснення файлів досягається за порядків моделі PPM в межах 4-12 для моделей, що не застосовують техніки спадкування інформації (порядків 16-32 інакше), тому при виборі деякого фіксованого порядку моделі втрачаємо у степені стиснення або використовуємо доволі багато ресурсів ЕОМ; у загальному випадку недостатньо добре відбуваєтья стиснення файлів, статистичні характеристики яких мають часті такі зміни, що оцінки розподілів ймовірностей у контекстних моделях швидко застарівають (нестабільність статистик контекстів); надто великі потреби пам'яті у випадку використання складних моделей високого порядку у сполученні з симетричністю алгоритму є препоною організації ефективного доступу до стиснених даних; програш в ефективності у порівнянні з алгоритмами типу LZ77 при стисненні файлів, що мають довгі повторювані блоки символів [7].

Практично завжди можна підібрати й налаштувати таку PPM-модель (контекстну модель з неявним зважуванням), що вона буде давати краще стиснення, ніж LZ. Але застосування PPM компресорів доцільне для стиснення текстів на природніх мовах та подібних до них даних, оскільки при обробці малонадлишкових файлів великі часові витрати. Надлишкові файли з довгими повторюваними рядками (наприклад, тексти програм) має смисл стискувати за допомогою інших компресорів, де співвідношення стиснення-швидкість-пам'ять кращі. Для дуже надлишкових даних краще використовувати PPM, так як методи LZ працюють при цьому порівняно повільно внаслідок деградації структур даних [7].

Основні характеристики алгоритмів родини PPM. Степені стиснення визначаються даними, для текстів звичайно 3-4. Алгоритми універсальні, але найкраще підходять для стиснення текстів. Симетричність: близька до 1; зазвичай декодер трохи повільніше кодера. Повільна обробка малонадлишкових даних [7].

Лекція 11. Алгоритми стиснення зображень – LZW, Хафмана

Розглянемо спершу *методи обходу площини*, оскільки відповідна задача виникає при обробці двовимірних даних (зображення) при створенні одновимірного масиву з двовимірного, який потім і буде стискуватися. Новий масив бажано створювати так, щоб «розривів» було як можна менше, тобто, щоб кожен наступний елемент, що заноситься до масиву D на і-му кроці, був сусіднім (у площині) для попереднього, занесеного до D на (і-1)-му кроці [6-7].

Змійка (зигзаг-сканування). Обхід масиву S починається з одного кутку площини, закінчується у протилежному по діагоналі. Вигідний коли в одному з кутків є "особливість" (наприклад, зосереджені найбільші коефіцієнти). Метод застосовується в алгоритмі JPEG для обходу квадрантів (розміром 8х8 точок).

Обхід рядками. Найпростіший метод, використовується в поширених графічних форматах (BMP, TGA, RAS тощо) для зберігання елементів зображень. У варіанті для рядків з розворотами для кожного другого рядку робиться вибірка у зворотному напрямі, тому тут точок «розриву» немає на відміну від звичайного підходу. Аналогічним чином може виконуватися обхід стовпцями.

Обхід смугами. У випадку обходу рядками поняття "області" відсутнє – кожен елемент вважають "областю". Намагаючись обходити площину квадратами разміром NxN, приходимо до ідеї обходу горизонтальними "смугами" шириною N:

1	4	7	10	13
2	5	8	11	14
3	6	9	12	15
16	19	22	25	28
17	20	23	26	29
18	21	24	27	30

а) просто смугами

1	6	7	12	13
2	5	8	11	14
3	4	9	10	15
28	27	22	21	16
29	26	23	20	17
30	25	24	19	18

б) смугами з розворотами

Обхід смугами з розворотами вже не дає точок «розриву», крім того кожна точка належить до області, записаної в D компактно, без розривів: її елементи розташовані у одному інтервалі (D[i], D[i+1], ..., D[i+j]) й елементів з інших областей у цьому інтервалі немає (наприклад, область – кожен з 4-х кутів розміром 3х3 елементи).

Обхід решітками. Для першої порції беруть елементи з кожного N-го стовпця кожного М-го рядку. Для другої – те ж, але із зсувом на 1

стовпчик. Так же й для наступних, а потім – із зсувом на 1, 2, (M-1) рядки. Наприклад, якщо M=N=2, то маємо 4 порції:

1	13	2	14	3	15	4	16
25	37	26	38	27	39	28	40
5	17	6	18	7	19	8	20
29	41	30	42	31	43	32	44
9	21	10	22	11	23	12	24
33	45	34	46	35	47	36	48

Бачимо, площина розбивається на прямокутники розміру MxN, задається обхід площини прямокутниками та обхід у самих прямокутниках й далі робиться "одночасний" обхід по кожному з них: спочатку вибираються їх перші елементи, потім другі, треті, й до останнього.

Обхід решітками з урахуванням значень елементів. Після того як оброблені перші елементи прямокутників (в прикладі – квадратів 2x2), якщо припустити, що вони є атрибутами своїх областей, вигідно (для покращення стиснення) згрупувати області з однаковими атрибутами, тобто з однаковими значеннями цих перших елементів. Нехай атрибути розподілені так:

R		G		G		G	
L		R		G		G	
L		L		R		R	

R	13	G	25	G	28	G	31
15	14	27	26	30	29	33	32
L	40	R	16	G	34	G	37
42	41	18	17	36	35	39	38
L	43	L	46	R	19	R	22
45	44	48	47	21	20	24	23

тоді має смисл спочатку обходити квадрати з атрибутом "R", потім з атрибутом "G", а далі – "L".

Контурний обхід. Нехай частина елементів належить до одної групи, частина – до іншої, причому контур їх розташування заданий. Маємо 36 елементів з групи "1" та 12 – з групи "2". Очевидно, має смисл окремо оформити елементи групи "1", а потім елементи групи "2".

1	1	1	1	1	1	1	1
1	1	2	2	1	1	1	1
1	2	2	2	2	1	1	1
1	2	2	2	2	2	1	1
1	1	1	2	1	1	1	1
1	1	1	1	1	1	1	1

а) розташування елементів груп

1	29	28	27	26	22	21	31
2	30			25	23	20	32
3					24	19	33
4						18	34
5	8	9		13	14	17	35
6	7	10	11	12	15	16	36

б) оформлення елементів окремо по групах

Контурний обхід з невідомими контурами. Нехай маємо попередній розподіл елементів груп по площині, але від початку обходу площини цей розподіл невідомий. Тоді можна діяти так: обходимо площину "стовпцями з розворотами", виявляючи елемент іншої групи (в елементах 9, 10, 14...), також робимо розворот на 180°. Далі (кроки 45-48) обходимо частину площини, що залишилася й містить (можливо) елементи другої групи (вар.1). В результаті:

• серед перших 36 елементів 4 з групи "2", а 32 з групи "1";

• з останніх 12 елементів 8 з групи "2", а 4 з групи "1".

В першій частині 4/36=1/9 "виключень", у другій їх – 4/12=1/3.

Якби виконували просто обхід "стовпцями з розворотами" (вар. 2), мали б:

1	44	41	40	35	34	29	28
2	43	42	39	36	33	30	27
3	45	46	38	37	32	31	26
4	9	10	47	48	19	20	25
5	8	11	14	15	18	21	24
6	7	12	13	16	17	22	23

а) варіант 1

1	12	13	24	25	36	37	48
2	11	14	23	26	35	38	47
3	10	15	22	27	34	39	46
4	9	16	21	28	33	40	45
5	8	17	20	29	32	41	44
6	7	18	19	30	31	42	43

б) варіант 2

серед перших 33 елементів 12 з групи "2", а 21 з групи "1" та з останніх 15 елементів всі з групи "1".

"Квадратна змійка". Рекурсивний метод для квадратних областей. Якщо прийняти лівий верхній елемент за перший, то для квадрата 2x2 можливі дв варіанти обходу без розривів:

1	4
2	3

1	2
4	3

.

У першому випадку п’ятим кроком буде перехід до лівого верхнього елементу квадрату справа або до правого нижнього елементу квадрату зверху. У другому – п’ятим кроком буде перехід до квадрату знизу або зліва. Вкажемо:

• якщо потрібно вийти до правого (верхнього) квадрату, то перший крок – униз, якщо до нижнього (лівого), то перший крок – вправо;

• шлях, що пройшли, однозначно задає до якого квадрату потрібно вийти в кожен конкретний момент;

• тільки на початку є вибір одного з двох варіантів обходу.

Тепер, якщо, наприклад, перший крок був у квадраті 2x2 вправо, то в результаті обходу його маємо вийти до нижнього квадрату 2x2. Перший крок переходу від 2x2 до 2x2 в 4x4 був униз, тому маємо вийти до

правого 4х4. Перший перехід у квадраті 16х16 був вправо, тому в результаті обходу 16х16 повинні прийти до нижнього. І т.д. Розривів не буде й кожен елемент належатиме до області, записаної компактно. Для квадрату 3х3 можемо узяти такі два "шаблони":

1	4	5
2	3	6
9	8	7

1	2	9
4	3	8
5	6	7

.

Перше правило буде "протилежним" першому правилу випадку 2х2, але останні два – такі ж самі:

• якщо потрібно вийти до правого (верхнього) квадрату, то перший крок – вправо, якщо до нижнього (лівого), то перший крок – униз;

• шлях, що пройшли, однозначно задає до якого квадрату потрібно вийти в кожен конкретний момент;

• тільки на початку є вибір одного з двох варіантів обходу.

Варіанти обходу, що залишилися, є еквівалентними, взаємозамінними, оскільки в обох випадках будемо виходити у правий нижній кут.

Аналогічно розглядаємо квадрати 5х5, 25х25 і т.д.

Якщо потрібно обійти квадрат розміру NxN, спочатку визначимо, добутком яких простих чисел є N, потім задаємо порядок цих чисел у добутку (а для кожного непарного множника ще й напрям обходу). Таким чином буде заданий процес обходу.

Якщо К, рівне числу двійок у добутку, більше одного: К>1, то сторона самого малого квадрату має бути 2^{K-1} або 2^{K} елементів. Наприклад, якщо К=3, то обхід без розривів 2х3х2х2 (від самого малого 2х2 до 6х6, потім до 12х12 й, накінець, до 24х24) неможливий. Інших обмежень при обході "квадратною змійкою" немає.

Кожен квадрат із стороною L > 2 можна обходити звичайною змійкою, а не "квадратною". Це вигідно тоді, коли елементи, що найбільш різняться, згруповані у протилежних кутках квадрату.

Обхід по спіралі. Відбувається просто, наприклад, будується квадрат 3х3, потім 5х5, 7х7, 9х9 і т.д. Спіраль може й сходитися (протилежний рух, від 49 до 1. Крім того, вона може бути з розворотами. Напрям змінюється в точках, розташованих на діагоналі, в прикладі це "25" і "41".

43	44	45	46	47	48	49
42	21	22	23	24	25	26
41	20	7	8	9	10	27
40	19	6	1	2	11	28
39	18	5	4	3	12	29
38	17	16	15	14	13	30
37	36	35	34	33	32	31

а) проста спіраль

1	2	3	4	5	6	7
24	25	40	39	38	37	8
23	26	41	42	43	36	9
22	27	48	49	44	35	10
21	28	47	46	45	34	11
20	29	30	31	32	33	12
19	18	17	16	15	14	13

б) спіраль з розворотами

Загальні моменти для прямокутних методів [6, 7]. Завжди можна почати з одного з чотирьох кутів й рухатися в одному з двох напрямків: по вертикалі й по горизонталі. Положення першого кута впливу на степінь стиснення майже не має, особливо при стисненні зображень. А вибір направлення може суттєво покращувати стиснення, оскільки області відносно основного направлення по вертикалі або по горизонталі будуть згруповані по-різному (наприклад, відсканований текст краще стискати, обходячи по вертикалі, оскільки в ньому більше довгих суцільних вертикальних ліній, ніж горизонтальних).

Загальні моменти для методів складної форми [6, 7]. Може виникнути необхідність помічати вже оброблені точки площини, щоб уникнути лишніх обчислень, попереджаючих повторне їх занесення до масиву D. На практиці існує є два основні варіанти: додати по одному "прапорцевому" біту для кожної точки площини або вибрати (додати) значення для "прапорця", що вказує, що точку вже занесено до масиву D, й записувати це значення на місце вже внесених точок.

Тепер перейдемо до розгляду алгоритмів стиснення зображень.

Алгоритми родини LZ

Існує досить велика родина LZ-подібних алгоритмів, що різняться пошуком повторюваних ланцюжків. За простих варіантів відповідного алгоритму може бути припущення, що у вихідному потоці йде пара <довжина співпадіння, зміщення відносно поточної позиції> або <довжина співпадіння> байтів, що «пропускаються», та самі значення байтів. При розтисненні для пари < довжина співпадіння, зміщення...> копіюються <довжина співпадіння> байтів з вихідного масиву, отриманого в результаті розтиснення, на <зміщення> байтів раніше, а <довжина співпадіння> (число, що дорівнює довжині співпадіння) значень байтів, що «пропускаються», просто копіюються у вихідний масив із вхідного потоку. Цей алгоритм є несиметричним за часом, оскільки вимагає повного перебору буферу (ковзаючого вікна) при

пошуку однакових підрядків. Внаслідок цього є складним задати великий буфер з причини різкого зростання часу стиснення. Але потенційно побудова алгоритму, в якому на <довжина співпадіння> й на <зміщення> буде виділено по 2 байти (старший біт старшого байту довжини співпадіння – ознака повтору рядку / копіювання потоку), дає можливість стискати всі повторювані підрядки розміром до 32Кб у буфері розміром 64Кб.

В результаті одержимо зростання розміру файлу у гіршому випадку на 32770/32768 (у 2 байтах записано, що потрібно переписати у вихідний потік наступні 215 байтів), що зовсім непогано. Границею максимальної степіні стиснення є 8192 рази (це граничне значення, оскільки максимальне стиснення одержимо перетворюючи 32 Кб буферу в 4 байти, а буфер ("словник" в термінах LZ) такого розміру буде накопичений не одразу. Але мінімальний підрядок, для якого нам вигідно проводити стиснення, має бути у загальному випадку мінімум з 5 байтів, що й визначає малу цінність даного алгоритму. Перевагою такого варіанту LZ є простота алгоритму декомпресії.

Алгоритм LZW (Lempel, Ziv и Welch)

Стиснення в алгоритмі здійснюється за рахунок однакових ланцюгів байтів. Для представлення та зберігання ланцюгів (фраз словника) використовується дерево, що доволі сильно обмежує вигляд ланцюгів і тому не всі однакові підланцюги у зображенні будуть використані при стисненні. Але в алгоритмі вигідно стискувати навіть ланцюги, що складаються з 2 байт.

Процес стиснення є доволі простим. Спочатку зчитують послідовно символи вхідного потоку й перевіряють, чи є у створеній таблиці рядків такий рядок. Якщо є, то зчитують наступний символ, якщо немає – заносять у потік код для попереднього знайденого рядку, вносять рядок до таблиці й починається пошук знову (в представленому нижче прикладі алгоритму використано функцію InitTable(), що очищує таблицю та розташовує у ній всі рядки одиничної довжини).

Функція InitTable() ініціалізує таблицю рядків так, щоб вона містила всі можливі рядки, що складаються з одного символу. Наприклад, якщо стискають байтові дані, то рядків у таблиці буде 256 ("0", "1", ..., "255"). Для коду очистки (ClearCode) і коду кінця інформації (CodeEndOfInformation) зарезервовані значення 256 і 257. У наведеному варіанті алгоритму використовується 12-бітовий код, і тому під

коди для рядків залишаються значення від 258 до 4095. Рядки, що додаються, записуються у таблицю послідовно, при цьому індекс рядку в таблиці стає її кодом. Функція ReadNextByte() зчитує символ з файлу, функція WriteCode() записує код (не рівний по розміру байту) у вихідний файл. Функція AddStringToTable () додає новий рядок до таблиці, приписуючи йому код, та оброблює ситуацію переповнення таблиці. У випадку переповнення у потік записується код попереднього знайденого рядка і код очищення, після чого таблиця очищується функцією InitTable(). Функція CodeForString() знаходить рядок у таблиці та видає код цього рядку.

```
InitTable()
CompressedFile.WriteCode(ClearCode)
CurStr = пустий рядок
while (не досягнемо кінця файлу ImageFile)
      c = ImageFile. ReadNextByte()
      if  (CurStr + c  є в таблиці)
          CurStr = CurStr + c // приклеювання символу до рядку
      else
          code = CodeForString(CurSt)
          CompressedFile.WriteCode(code)
          AddStringToTable(CurStr + c)
          CurStr = c  // рядок з одного символу
code = CodeForString(CurSt)
CompressedFile.WriteCode(code)
CompressedFile.WriteCode(CodeEndOfInformation)
```

Нехай стискаємо послідовність 45, 55, 55, 151, 55, 55, 55. За алгоритмом розташовуємо у вихідний потік спочатку код очищення <256>, потім додаємо до початково пустого рядку "45" та перевіряємо, чи є рядок "45" у таблиці. Оскільки при ініціалізації занесли у таблицю всі рядки довжиною в один символ, то рядок "45" є в таблиці. Далі читаємо наступний символ 55 з вхідного потоку та перевіряємо, чи є рядок "45, 55" у таблиці. Такого рядку у таблиці поки що немає. Заносимо до таблиці рядок "45, 55" (з першим вільним кодом 258) та записуємо у потік код <45>. Коротко процес стиснення можна представити так:
"45" – є в таблиці;
"45, 55" – немає. Додаємо у таблицю <258>"45, 55". В потік: <45>;
"55, 55" – немає. В таблицю: <259>"55,55". В потік: <55>;

"55,151" – немає. В таблицю: <260>"55, 151". В потік: <55>;
"151,55" – немає. В таблицю: <261>"151,55". В потік: <151>;
"55, 55" – є в таблиці;
"55,55,55" - немає. В таблицю: "55,55, 55" <262>. В потік: <259>.
Послідовність кодів для прикладу, що попадає у вихідний потік: <256>, <45>, <55>, <55>, <151>, <259>.
Особливість алгоритму LZW: для декомпресії не потрібно зберігати таблицю рядків у файлі для розтиснення, можна відновити таблицю рядків, користуючись тільки потоком кодів (відомо, що для кожного коду потрібно додавати в таблицю рядок, що складається з вже присутнього там рядку та символу, з якого починається наступний рядок у потоці).
Алгоритм декомпресії може виглядати так [7]:

```
code = File.ReadCode( )
while (code є CodeEndOfInformation)
      if  (code == ClearCode)
           InitTable()
           code = File.ReadCode( )
      if  (code == CodeEndOfInformation)
           закінчити роботу
          else
               if  (InTable(code))
                    ImageFile.WriteString(FromTable(code))
                    AddStringToTable(StrFromTable(old_code)+
                                FirstChar(StrFromTable (code)))
                    old_code = code
                else
                     OutString = StrFromTable(old_code)+
                       FirstChar(StrFromTable(old_code))
                     ImageFile.WriteString(OutString)
                     AddStringToTable(OutString)
                     old_code = code
          code = File.ReadCode( )
```

В представленому алгоритмі функція ReadCode() зчитує черговий код з розтискуваного файлу. Функція InitTable() як і раніше (при стисненні) очищує таблицю й заносить до неї всі рядки з одного символу. Функція FirstChar() видає перший символ рядку. Функция StrFromTable() видає

рядок з таблиці по коду. Функція AddStringToTable() додає новий рядок у таблицю (надає їй перший вільний код). Функція WriteString() записує рядок у файл.

В даному алгоритмі записувані у потік коди поступово зростають, доки у таблиці не з'явиться, наприклад, вперше код 512, всі коди будуть меньші 512. При стисненні та при розтисненні коди у таблиці додаються при обробці одного й того ж символу, тобто дія відбувається "синхронно". Можна скористатися цією властивістю алгоритму, щоб підвищити степінь компресії. Поки у таблицю не доданий 512-й символ, пишемо у вихідний бітовий потік код з 9 бітів, а при додаванні 512 – коди з 10 бітів. Відповідно декомпрессор також має сприймати всі коди вхідного потоку 9-бітовими до моменту додавання у таблицю коду 512, після чого буде сприймати всі вхідні коди як 10-бітові. Аналогічно відбуваються зміни довжини при додаванні у таблицю кодів 1024 та 2048 (11 та 12 біт, відповідно). Такий підхід дозволяє приблизно на 15% підвищити степінь компресії:

При стисненні зображення важливо забезпечити швидкість пошуку рядків у таблиці. Можна скористатися тим, що кожен наступний підрядок на один символ за довжиною більше попереднього, крім того, попередній рядок вже був знайдений у таблиці. Тому достатньо створити список посилань на рядки, що починаються з даного підрядка, й процес пошуку в таблиці зведеться до пошуку у рядках, що містяться у списку для попереднього рядку, а це можна виконати дуже швидко.

На практиці достатоньо зберігати у таблиці тільки пару <код попереднього підрядку, доданий символ>. Цієї інформації достатньо для роботи алгоритму. Отже, масив від 0 до 4095 з елементами <код попереднього підрядку; доданий символ; список посилань на рядки, що починаються з цього рядка> вирішує задачу пошуку, але дуже повільно.

На практиці для зберігання таблиці використовують хеш-таблиці. Таблиця складається з 2^{13} елементів. Кожен елемент містить <код попереднього підрядку; доданий символ; код цього рядку>. Ключ для пошуку довжиною 20 біт формується з використанням двох перших елементів, що зберігаються у таблиці як одно число (key). Молодші 12 бітів цього числа надані під код, а старші 8 бітів – під значення символу.

Найкраще стиснення даним алгоритмом одержимо для ланцюга з однакових байтів великої довжини (тобто для 8-бітового зображення, всі точки якого мають, для визначеності, колір 0). При цьому в 258 рядках

таблиці запишемо рядок "0, 0", у 259 – "0, 0, 0", ... у 4095 – рядок з 3839 (= 4095-256) нулів, а у потік попаде 3840 кодів, включаючи код очищення. Звідси, знайшовши суму арифметичної прогресії від 2 до 3839 (довжину стисненого ланцюга) й поділивши її на 3840*12/8 (у потік записуються 12-бітові коди), має найкращу степінь стиснення.

Найгірше стиснення отримаємо, якщо жодного разу не зустрінемо підрядок, що вже є в таблиці (в ній не повинно зустрітися жодної пари однакових символів). Якщо постійно будемо зустрічати новий підрядок, то запишемо у вихідний потік 3840 кодів, яким буде відповідати рядок з 3838 символів. Без урахування зміни довжини кодів все це складе збільшення файлу майже у 1.5 рази.

Алгоритм LZW реалізований у форматах GIF та TIFF.

Основні характеристики алгоритму LZW [7]. Степені стиснення: найкраща – приблизно 1000, середня – 4, найгірша – 5/7. Стиснення відбувається за рахунок однакових підланцюгів у потоці. Стиснення у 1000 разів досягається лише на однокольорових зображеннях розміром кратним приблизно 7 Мб. Алгоритм орієнтований на 8-бітові зображення, побудовані на комп'ютері. Алгоритм майже симетричний за умови оптимальної реалізації операції пошуку рядку в таблиці. Ситуації, коли алгоритм збільшує розміри, зустрічаються дуже рідко.

Алгоритм Хафмана

Алгоритм Хафмана з фіксованою таблицею [6, 7]. Класичний алгоритм практично не застосовується до зображень у чистому вигляді, його використовують як один з етапів компресії у більш складних схемах. Модифікація алгоритму використовується при стисненні чорно-білих зображень (1 біт на піксел). Назва відповідного алгоритму: CCITT Group 3 (запропонований 3 групою із стандартизації Consultative Committee International Telegraph and Telephone). В алгоритмі послідовності чорних і білих точок, що йдуть підряд, замінюють числом, рівним їх кількості. Одержаний ряд потім стискується за Хафманом із використанням фіксованої таблиці. Кожен рядок зображення в алгоритмі стискується незалежно.

Набір точок зображення одного кольору, що йдуть підряд, називають **серією**. Довжину цього набору точок називають **довжиною серії**. У фіксованій таблиці задані два види кодів:

• коди завершення серій – задані від 0 до 63 з кроком 1;

• складені (додаткові) коди – задані від 64 до 2560 з кроком 64.

Вважається, що у зображенні суттєво переважає білий колір, й усі рядки зображення починаються з білої точки. Якщо рядок починається з чорної точки, то вважається, що рядок починається білою серією довжини 0. Наприклад, послідовність довжин серій 0, 3, 556, 10,... означає, що в рядку зображення йдуть спочатку 3 чорні точки, потім 556 білих, далі 10 чорніх і т. д. На практиці у випадках, коли в зображенні переважає чорний колір, інвертують зображення перед компресією й записують інформацію про це у заголовок файлу.
Алгоритм компресії можна записати так:

```
for (для всіх рядків зображення)
    Перетворити рядок у множину довжин серій
    for (для всіх серій)
        if (серія біла)
            L = довжина серії
            while (L > 2623)  //2560+63
                L = L – 2560
                WriteWhiteCodeFor(2560)
             if (L > 63)
                L2 = MaxComposCodeLessL(L)
                L = L – L2
                WriteWhiteCodeFor(L2)
            WriteWhiteCodeFor(L) // буде кодом завершення
            code = CodeForString(CurSt)
            else
                [Аналогічно для чорних кодів]
    end for
end for
```

Оскільки чорні й білі серії чергуються, то код для білої й код для чорної серій працюють по черзі. В термінах регулярних виразів маємо для кожного рядку зображення (є достатньо довгим, починається з білої точки) вихідний бітовий потік виду:
((<Б-2560>*[<Б-сст.>]<Б-зв.>(<Ч-2560>)*[<Ч-сст.>]<Ч-зв>)$^{+}$
[(< Б-2560>)*[<Б-сст.>]<Б-зв.>],
де ()* – повторювання 0 або більше разів; ()+ – повторювання 1 або більше разів; [] – включення 1 або 0 разів.

Для наведеної раніше послідовності 0, 3, 556, 10... алгоритм сформує код: <Б-0><Ч-3><Б-512><Б-44><Ч-10>…, або за таблицею 00110101100110010*00101101***0000100**.

Отриманий код є префіксним й легко може бути згорнутий знову у послідовність довжин серій. Для наведеного рядку у 569 бітів отримали код довжиною в 33 біти, отже степінь стиснення складає приблизно 17 разів.

Найскладнішим виразом у представленому вище алгоритмі є L2 = MaxComposCodeLessL(L), що працює так: L2 = (L»6)*64, де "»" – побітовий зсув L вліво на 6 бітів.

Таблиця для практичного використання алгоритму побудована за допомогою класичного алгоритму Хафмана (окремо для довжин чорних і білих серій), в ній значення ймовірностей появи для конкретних довжин серій отримані шляхом аналізу великої кількості факсимільних зображень (табл. 11.1).

Алгоритм Хафмана з фіксованою таблицею використовують у форматі TIFF.

Основні характеристики алгоритму CCITT Group 3 [7]. Степені стиснення: найкраща – прямує до граничного значення 213.(3), середня – 2, у найгіршому випадку файл зростає у 5 разів. Клас зображень: двокольорові чорно-білі зображення, в яких переважають великі простори, заповнені білим кольором. Симетричність близька до 1. Алгоритм дуже простий в реалізації, швидкий, може бути легко реалізованим апаратно.

Адаптивний алгоритм Хафмана [6-7, 9]. Також використовується як один з етапів компресії у більш складних схемах. В алгоритмі кожен рядок зображення стискається незалежно. Є модифікацією класичного алгоритму (де для збору статистики маємо два проходи), але дозволяє не передавати таблицю кодів та обмежитися одним проходом по рядку при кодуванні (а також при декодуванні).

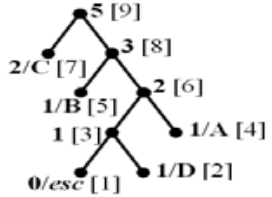
В даній модифікації алгоритму (при кодуванні або декодуванні) при кожному співставленні символу коду змінюється внутрішній код обчислень так, що наступного разу для цього ж символу може бути наданий інший код внаслідок адаптації алгоритму до символів, що надходять для кодування.

В алгоритмі використовується впорядковане бінарне дерево (його вузли можуть бути перераховані в порядку незростання їхньої ваги), перелічення вузлів відбувається по ярусах зверху до низу та зліва направо

Таблиця 11.1. Коди завершення алгоритму [7].

Довжина серії	Код білого підрядку	Код чорного підрядку	Довжина серії	Код білого підрядку	Код чорного підрядку
0	00110101	0000110111	32	00011011	000001101010
1	00111	010	33	00010010	000001101011
2	0111	11	34	00010011	000011010010
3	1000	10	35	00010100	000011010011
4	1011	011	36	00010101	000011010100
5	1100	0011	37	00010110	000011010101
6	1110	0010	38	00010111	000011010110
7	1111	00011	39	00101000	000011010111
8	10011	000101	40	00101001	000001101100
9	10100	000100	41	00101010	000001101101
10	00111	0000100	42	00101011	000011011010
11	01000	0000101	43	00101100	000011011011
12	001000	0000111	44	00101101	000001010100
13	000011	00000100	45	00000100	000001010101
14	110100	00000111	46	00000101	000001010110
15	110101	000011000	47	00001010	000001010111
16	101010	0000010111	48	00001011	000001100100
17	101011	0000011000	49	01010010	000001100101
18	0100111	0000001000	50	01010011	000001010010
19	0001100	00001100111	51	01010100	000001010011
20	0001000	00001101000	52	01010101	000000100100
21	0010111	00001101100	53	00100100	000000110111
22	0000011	00000110111	54	00100101	000000111000
23	0000100	00000101000	55	01011000	000000100111
24	0101000	00000010111	56	01011001	000000101000
25	0101011	00000011000	57	01011010	000001011000
26	0010011	000011001010	58	01011011	000001011001
27	0100100	000011001011	59	01001010	000000101011
28	0011000	000011001100	60	01001011	000000101100
29	00000010	000011001101	61	00110010	000001011010
30	00000011	000001101000	62	00110011	000001100110
31	00011010	000001101001	63	00110100	000001100111
64	11011	0000001111	1344	011011010	0000001010011
128	10010	000011001000	1408	011011011	0000001010100
192	01011	000011001001	1472	010011000	0000001010101
256	0110111	000001011011	1536	010011001	0000001011010
320	00110110	000000110011	1600	010011010	0000001011011
384	00110111	000000110100	1664	011000	0000001100100
448	01100100	000000110101	1728	010011011	0000001100101
512	01100101	0000001101100	1792	00000001000	совп. с белой
576	01101000	0000001101101	1856	00000001100	– // –
640	01100111	0000001001010	1920	00000001101	– // –
704	011001100	0000001001011	1984	000000010010	– // –
768	011001101	0000001001100	2048	000000010011	– // –
832	011010010	0000001001101	2112	000000010100	– // –
896	011010011	0000001110010	2176	000000010101	– // –
960	011010100	0000001110011	2240	000000010110	– // –
1024	011010101	0000001110100	2304	000000010111	– // –
1088	011010110	0000001110101	2368	000000011100	– // –
1152	011010111	0000001110110	2432	000000011101	– // –
1216	011011000	0000001110111	2496	000000011110	– // –
1280	011011001	0000001010010	2560	000000011111	– // –

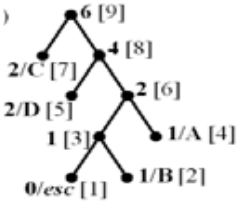
у кожному ярусі (наприклад, як на рис. 11.1, де esc – спеціальний символ, який завжди має частоту 0 й є необхідним для занесення до дерева нових символів).



Номер вузла	1	2	3	4	5	6	7	8	9
Вага вузла	0	1	1	1	1	2	2	3	5

Рис. 11.1 Впорядковане бінарне дерево алгоритму

На початку роботи алгоритму дерево кодування містить лише один спеціальний символ <**esc**>. Ліві гілки дерева помічають 0, праві – 1. При порушенні впорядкованості дерева (наприклад, після додавання нового листка або зміни ваги деякого листка) його потрібно впорядковувати. Для цього необхідно поміняти місцями два вузли – вузол, вага якого порушила впорядкованість, та останній з наступних (за цим вузлом) вузлів меншої ваги, а потім перелічити ваги всіх їх вузлів-пращурів. Наприклад, якщо на вхід поступив символ D, а за попередніми надходженнями символів вже отримали дерево, наведене вище, то вага листка D зросте на 1 та стане рівною 2 й порушиться впорядкованість (послідовність ваг не буде вже неспадаючою). Тому необхідно поміняти місцями лист D, який порушив впорядкованість, та лист B, що є останнім при переліченні з листів з меншою вагою. В результаті отримаємо таке дерево:



При кодуванні елементи на вході зчитуються по байтам. Якщо вхідний символ присутній у дереві, то у вихідний потік записується код, що

відповідає послідовності нулів та 1, якими помічені гілки дерева, при проході від кореня дерева до даного листку. Вага даного листка зростає на 1. Ваги вузлів-пращурів корегуються. Якщо дерево стає невпорядкованим, то відбувається його впорядкування. Якщо черговий символ, що надійшов, відсутній у дереві, у вихідний потік записується набір нулів та одиниць, якими помічені гілки дерева при русі від кореня до символу <**esc**>, а потім 8 бітів ASCII-коду нового символу. В дерево замість символу <**esc**> додається нова гілка: батько та 2 нащадки, де лівий нащадок є символом <**esc**>, а правий – новий доданий символ. Ваги вузлів пращурів корегуються, а дерево за необхідності впорядковується.

Нехай, на вході були символи A, B, за якими побудували дерево, наведене на рис. 11.2 (а). Далі надходить символ C, якого ще немає в дереві. Тому вихідний код нового символу буде 00'C' ('C' – ASCII-код символу C). До дерева додається гілка з новим символом (рис. 11.2, б) та дерево впорядковується (рис. 11.2, в).

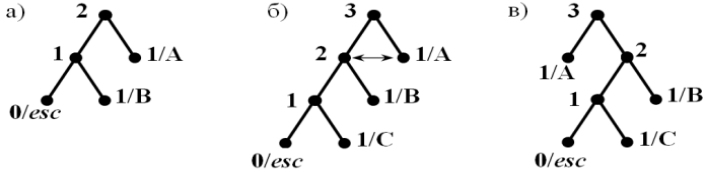
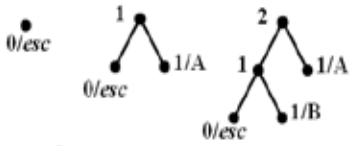


Рис. 11.2 Побудова дерева в процесі надходження символів

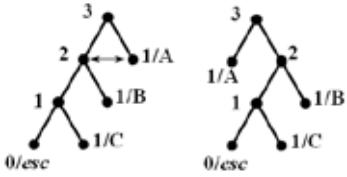
Розглянемо процес кодування для простого прикладу з 10 символів ABCCDDDDBB [6]. Результати занесемо до таблиці:

Вхідні дані	Вихідні дані	№ дерева
A	'A'	1
B	0'B'	2
C	00'C'	3
C	101	4
D	100'D'	5
D	1101	6
D	10	7
D	0	8
B	1101	9
B	111	10

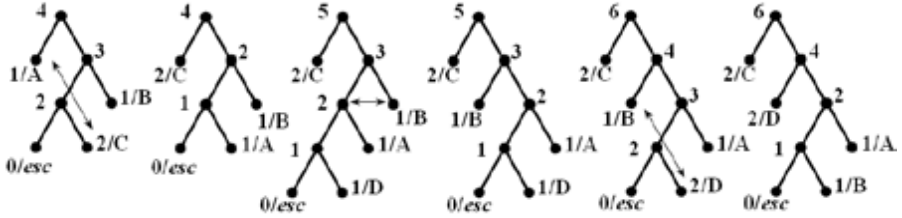
Побудова дерева рядку відбувається наступним чином. Спочатку надходять перші два символи.



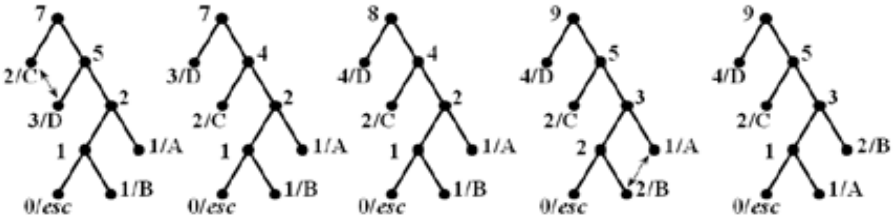
Після надходження третього символу (c) необхідне впорядкування бінарного дерева.



Далі надходить ще один символ C, тому вага відповідного вузла змінюється, дерево стає невпорядкованим, відбувається його впорядкування. Починають надходити символи D. З першим символом D додається вузол, проводиться впорядкування бінарного дерева, з надходженням другого символу D дерево стає невпорядкованим, відбувається його впорядкування.



З надходженням третього символу D дерево знову стає невпорядкованим, відбувається впорядкування. Надходження четвертого символу D не викликає потреби впорядкування. Далі надходить символ B, який змінює вагу вузла на 2, дерево стає невпорядкованим, відбувається його впорядкування.



При надходженні останнього, десятого, символу (є символом третім B) знову стає невпорядкованим, відбувається впорядкування.

Маємо, послідовність ABCCDDDDBB займає 80 бітів. Стиснені дані 'A'0'B'00'C'101100'D'11011001101111 вже займають 23 + 4*8 = 55 бітів. При декодуванні елементи також зчитуються по байтам. Кожен раз при зчитуванні 0 або 1 відбувається переміщення від кореня униз по відповідній гілці дерева, доки не буде досягнутий лист дерева. Якщо лист, що відповідає символу, досягнутий, то у вихідне повідомлення записується ASCII-код цього символу. Вага даного листка зростає на 1. Ваги вузлів-пращурів корегуються. Якщо дерево стає невпорядкованим, то відбувається його впорядкування. Якщо досягнутий символ <**esc**>, то з вхідного повідомлення зчитуються наступні 8 бітів, що відповідають ASCII-коду символу. До дерева додається новий символ, ваги вузлів-пращурів корегуються, а якщо дерево стає невпорядкованим, відбувається його впорядкування.

Наприклад, декодується повідомлення 'A'0'B'0100'C'11 у 31 біт. На початку дерево містить лише символ <**esc**> з частотою 0. Далі зчитуються символи, в процесі декодування будується та перебудовується дерево. В результаті одержимо послідовність ABBCBB.

Вхідні дані	Вихідні дані	№ дерева
'A'	A	1
0'B	B	2
01	B	3
00'C'	C	4
1	B	5
1	B	6

Основні характеристики алгоритму. Степені стиснення: середня – близько до 2, у найгіршому випадку файл зростає. Симетричність близька до 1. Алгоритм простий в реалізації, доволі швидкий.

Лекція 12. Стиснення інформації з втратами. Алгоритм JPEG. Ресурсна ефективність комп'ютерних алгоритмів

Стиснення інформації з втратами. Всі алгоритми стиснення, що розглядали, – алгоритми без втрат. Історично такі алгоритми були розроблені першими. Їх використовували (і зараз використовують) в системах резервного копіювання, при створенні дистрибутивів та ін. Але з поширенням нових класів зображень існуючі алгоритми перестали задовольняти вимогам, що висувають до стиснення, часто зображення практично не стискувалися, хоча й мали явну надлишковість. Все це стало причиною розвинення нового типу алгоритмів – стиснення із втратою інформації (де степінь стиснення та втрат якості можна задавати).

На сьогоднішній день не існує єдиного адекватного критерію оцінки втрат якості зображення, втрати ж при роботі з зображеннями відбуваються постійно (оцифрування, переведення в обмежену палітру кольорів або в іншу систему представлення кольорів для друку, при стисненні з втратами). В результаті використовують різні критерії, яким властиві суттєві недоліки.

Найпростіший критерій – *середнєквадратичне відхилення* значень пікселів (*L2 міра*, RMS), $d(x,y)=\sqrt{\frac{\sum_{i=1,j=1}^{n,n}(x_{ij}-y_{ij})^2}{n^2}}$. За ним зображення буде сильно зіпсоване при зменшенні яскравості всього на 5% (але око не відчує). В той же час зображення "із снігом" – різкою зміною кольору окремих точок, слабкими смугами або "муаром" – будуть визнані такими, що "майже не змінилися".

Критерій, *максимальне відхилення* – $d(x,y)=\max_{i,j}\left|x_{ij}-y_{ij}\right|$, як міра є дуже чуттєвим до биття окремих пікселів (в усьому зображенні може суттєво змінитися тільки значення 1 піксела, що практично не видно оку, але за такою мірою зображення буде вважатися сильно зіпсованим).

Міра відношення сигнала до шуму (PSNR), що поширена на практиці, – $d(x,y)=10\log_{10}\frac{255^2n^2}{\sum_{i=1,j=1}^{n,n}(x_{ij}-y_{ij})^2}$, по суті аналогічна середнєквадратичному

відхиленню. Користуватися нею зручніше за рахунок логарифмічного масштабу шкали, але має ті ж недоліки, що й середнєквадратичне відхилення.

Гарним вважають стиснення, за якого неможливо оком розрізнити початкове й розпаковане зображення. Непоганим – коли сказати, яке із зображень було стисненим, можна тільки порівнюючи дві картинки, що знаходяться поруч. При зростанні степені стиснення, як правило, стають примітними побічні ефекти, що характерні для обраного алгоритму. Крім того, навіть за відмінного збереження якості, до зображення можуть бути внесені регулярні специфічні зміни. Тому алгоритми стиснення з втратами не рекомендують використовувати при стисненні зображень, які збираються друкувати з високою якістю або обробляти програмами розпізнавання образів.

Розглянемо алгоритм стиснення зображень, який є алгоритмом стиснення із втратами, – алгоритм JPEG.

Алгоритм JPEG розроблений групою експертів в області фотографії для стиснення 24-бітових зображень (JPEG – Joint Photographic Expert Group – підрозділ в рамках ISO) [6, 7, 9]. Алгоритм JPEG практично є стандартом для повнокольорових зображень. Оперує областями 8x8, на яких яскравість та колір змінюються доволі плавно. Внаслідок цього при розкладенні матриці такої області у подвійний ряд за косінусами значущими виявляються тільки перші коефіцієнти. Стиснення в JPEG і відбувається за рахунок плавності зміни кольорів у зображенні.

Алгоритм базується на дискретному *косинусоїдальному перетворенні* (*ДКП*), що застосовується до матриці зображення для отримання деякої нової матриці коефіцієнтів. Для одержання початкового зображення застосовують обернене перетворення. ДКП розкладає зображення за амплітудами деяких частот. При перетворенні одержується матриця, в якій багато коефіцієнтів близькі або рівні нулю. Крім того, завдяки вадам людського зору можна апроксимувати коефіцієнти більш грубо без примітної втрати якості зображення. Для цього використовують квантування коефіцієнтів. У найпростішому випадку – це арифметичний побітовий зсув вправо. При цьому перетворенні втрачається частина інформації, але може досягатися більша степінь стиснення.

Нехай стискаємо 24-бітове зображення. На рис. 11.1 наведено послідовність дій алгоритму. Розглянемо їх більш докладно.

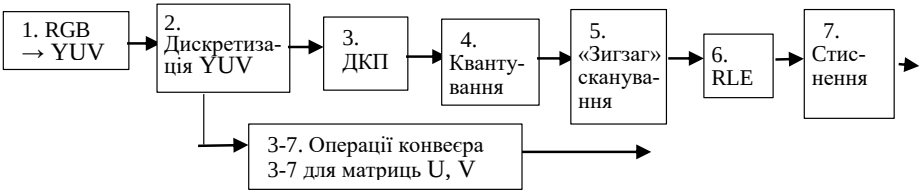


Рис. 11.1. Послідовність дій алгоритму JPEG

1. Переводимо зображення з кольорового простору RGB, з компонентами, що відповідають за червоний (Red), зелений (Green) і синій (Blue) складові кольора точки, у кольоровий простір YCrCb (іноді називають YUV), в якому Y – складова яскравості, а Cr, Cb – компоненти, що відповідають за колір (хроматичний червоний і хроматичний синій). За рахунок того, що око менш чуттєве до кольору, ніж до яскравості, маємо можливість архівувати масиви для Cr і Cb компонент з більшими втратами та, відповідно, із більшим стисненням.

Спрощено переведення з кольорового простору RGB у кольоровий простір YCrCb можна представити за допомогою матриці переходу:

$$\begin{pmatrix} Y \\ Cr \\ Cb \end{pmatrix} = \begin{pmatrix} 0.2990 & 0.5870 & 0.1140 \\ 0.5000 & -0.4187 & -0.0813 \\ -0.1687 & -0.3313 & 0.5000 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} + \begin{pmatrix} 0 \\ 128 \\ 128 \end{pmatrix}.$$

Обернене перетворення відбувається множенням вектора YUV на обернену матрицю.

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1.40200 \\ 1 & -0.34414 & -0.71414 \\ 1 & 1.77200 & 0 \end{pmatrix} \begin{pmatrix} Y \\ Cr-128 \\ Cb-128 \end{pmatrix}.$$

2. Розбиваємо початкове зображення на матриці 8x8. Формуємо з кожної 3 робочі матриці ДКП – по 8 біт окремо для кожної компоненти. При великих степенях стиснення цей крок може виконуватися дещо складніше. Зображення поділяють за компонентою Y, як і вище, а для компонент Cr і Cb матриці набираються через рядок і через стовпчик. Тобто з початкової матриці розміром 16x16 одержимо тільки одну робочу матрицю ДКП. При цьому втрачаємо 3/4 корисної інформації про кольорові складові зображення й одержуємо одразу стиснення у 2 рази.

На результуючому RGB-зображенні це відгукнеться несильно. Результат – стандартний формат YUV 4:1:1, який часто є вхідним для відеокодерів.

Стандарт не зобов'язує виконувати цю операцію, але такий підхід дозволяє підвищити ефективність стиснення. Далі початкове зображення розбивають на матрицю клітин однакового розміру (8×8 пікселів). Цей розмір вибраний з таких причин: з точки зору апаратної й програмної реалізації розмір блоку 8x8 не накладає суттєвих обмежень на розмір необхідної пам'яті; обчислювальна складність ДКП для блоку 8x8 також є прийнятною для більшості обчислювальних платформ.

3. У спрощеному виді ДКП при $n = 8$ можна представити:

$$C(i,u) = A(u)\cos\left(\frac{(2i+1)u\pi}{2n}\right),$$

$$Y[u,v] = \frac{1}{4}\sum_{i=0}^{7}\sum_{j=0}^{7} C(i,u)C(j,v)y[i,j], \quad A(u) = \begin{cases} \frac{1}{\sqrt{2}}, u = 0 \\ 1, u \neq 0 \end{cases}.$$

Застосовуємо ДКП до кожної робочої матриці. При цьому одержимо матрицю, в якій коефіцієнти в лівому верхньому кутку відповідають низькочастотній складовій зображення, а у правому нижньому – високочастотній. Поняття частоти випливає з розгляду зображення як дво вимірного сигналу. Плавна зміна кольору відповідає низькочастотній складовій, а різкі коливання – високочастотній.

4. Виконується квантування як ділення робочої матриці на матрицю квантування поелементно. Для кожної компоненти (Y, U і V) у загальному випадку задають свою *матрицю квантування* $q[u,v]$ (*МК*).

$$Yq[u,v] = IntegerRound\left(\frac{Y[u,v]}{q[u,v]}\right)$$

Елементи МК лінійно зростають пропорційно сумі індексів елементу матриці. Наприклад,

3	5	7	9	11	13	15	17
5	7	9	11	13	15	17	19
7	9	11	13	15	17	19	21
9	11	13	15	17	19	21	23
11	13	15	17	19	21	23	25
13	15	17	19	21	23	25	27
15	17	19	21	23	25	27	29
17	19	21	23	25	27	29	31

На цьому етапі відбувається керування степенем стиснення і отримують найбільші втрати. Задаючи МК з великими коефіцієнтами, одержимо більше нулів та більшу степінь стиснення. До стандарту JPEG включені рекомендовані МК, що побудовані емпірично. Матриці для більшої або меншої степені стиснення одержують множенням початкової матриці на деяке число gamma. З квантуванням пов'язані й специфічні ефекти алгоритму. За великих значень gamma втрати в нижніх частотах можуть бути настільки великими, що зображення розпадеться на квадрати 8x8. Втрати у високих частотах можуть проявитися і в ефекті Гібса (навколо контурів з різким переходом кольору утворюється "німб").

5. Переводимо матрицю 8x8 у 64-елементний вектор за допомогою "зигзаг"-сканування, тобто проходом елементів з індексами (0,0), (0,1), (1,0), (2,0), (1,1), (0,2), (0,3), (1,2), (2,1)(3,0), ..., (7,7). В результаті на початку вектору одержуємо коефіцієнти матриці, що відповідають низьким частотам, а на кінці – високим.

6. Згортаємо вектор за допомогою алгоритму групового кодування. Одержуємо пари типу <пропустити, число>, де "пропустити" є лічильником нулів, що пропускаємо, а "число" – значення, яке необхідно поставити у наступну комірку. Наприклад, вектор 42 3 0 0 0 -2 0 0 0 0 1 ... буде згорнутий у пари (0,42) (0,3) (3,-2) (4,1) ... .

7. Стискаємо отримані пари кодуванням по Хафману з фіксованою таблицею.

Процес відновлення зображення (розтиснення) в алгоритмі повністю симетричний. Обернене ДКП має вигляд:

$$y[u,v] = \frac{1}{4}\sum_{i=0}^{n}\sum_{j=0}^{n} C(i,u)C(j,v)Y[i,j],\ \ n = 7,$$

$$C(i,u) = A(u)\cos\left(\frac{(2i+1)u\pi}{2n}\right),\ \ A(u) = \begin{cases} \frac{1}{\sqrt{2}}, u = 0 \\ 1, u \neq 0 \end{cases},$$

Втрати при оберненому ДКП також не великі у порівнянні із втратами квантування.

Метод дозволяє стискати деякі зображення у 10-15 разів без суттєвих втрат. *Перевагами* алгоритму є: задання степіні стиснення; вихідне кольорове зображення може мати 24 біти на точку. Недоліками: за підвищення степені стиснення зображення розпадається на окремі квадрати (8x8) (пов'язане з тим, що відбуваються великі втрати у низьких час

тотах при квантуванні й відновити початкові дані стає неможливо); проявляється ефект Гібса.

Стандартизований JPEG у 1991 р. і вже тоді існували алгоритми, що стискали більше за менших втрат якості. Але розробникі враховували обмеження потужностей техніки, що існувала на той час. Їхні представлення були наступними: навіть на ПК алгоритм мав працювати менше хвилини на середньому зображенні, а його апаратна реалізація повинна бути відносно простою й дешевою, алгоритм має бути симетричним. Виконання цієї вимоги зробило можливим появу цифрових фотоапаратів, що знімають 24-бітові фотографії на 8-256 Мб флеш-карту. Якби алгоритм був несиметричним, мали б чекати, доки апарат "перезаряджується" – стисне зображення. Проблемою JPEG є те, що іноді горизонтальні й вертикальні смуги на екрані абсолютно не видимі й можуть проявитися тільки на друці у вигляді муарового візерунку. Він виникає при накладенні похилого растру друку на горизонтальні та вертикальні смуги зображення. Тому JPEG не рекомендують активно використовувати в поліграфії, задаючи високі коефіцієнти матриці квантування. Але при архівації зображень, призначених для перегляду людиною, він чудовий. Широке застосування JPEG тривалий час стримувалось тим, що він оперує 24-бітовими зображеннями. З цієї причини для того, щоб з прийнятною якістю переглянути картинку на звичайному моніторі у 256-кольоровій палітрі, потрібне застосування відповідних алгоритмів, а тому й деякий час.

JPEG є стандартом ISO, але формат його файлів не був зафіксованим. В результаті виробники створюють власні, несумісні між собою формати й змінюють алгоритм. Так, внутрішні таблиці алгоритму, рекомендовані ISO, замінюються ними на власні. Крім того, зміни відбуваються й при заданні степені втрат. Наприклад, при тестуванні бачать, що "відмінну" якість, "100%" і "10 балів" дають суттєво різні картинки. Існують варіанти JPEG для специфічних додатків. Як стандарт ISO JPEG все ширше використовують при обміні зображеннями у комп'ютерних мережах.

Основні характеристики алгоритму JPEG [7]. Степінь стиснення: 2-200 (задається користувачем). Призначений для повнокольорових 24-бітових зображень або зображень в градаціях сірого без різких переходів кольорів (фотографії). Алгоритм симметричний. Іноді алгоритм створює "німб" навколо різких горизонтальних і вертикальних границь у зображенні (ефект Гібса), за високої степені стиснення зображення розпадається на блоки 8x8 пікселів.

Ресурсна ефективність комп'ютерних алгоритмів. Використовується для порівняльного аналізу й раціонального вибору алгоритмів у реальному діапазоні довжин входів з метою підвищення ресурсної ефективності алгоритмічного забезпечення програмних засобів (спрямована на розробку методів оцінки й вибору раціональних алгоритмів рішення обчислювальних задач у заданих умовах застосування). Самими поширеними характеристиками ресурсної ефективності алгоритмів є оцінки часової та ємнісної складності, що відображують потрібні ресурси процесора і оперативної пам'яті та/або зовнішньої пам'яті.

Часова складність та функція працеємності. Класичний аналіз обчислювальних алгоритмів пов'язаний з аналізом їх часової складності. Його результатом є асимптотична оцінка кількості операцій, що задається алгоритмом, як функції довжини входу, що корельовано з асимптотичною оцінкою часу виконання програмної реалізації алгоритму. Але асимптотичні оцінки вказують лише порядок зростання функції, а тому результати порівняння алгоритмів за цими оцінками будуть виконуватися за дуже великих довжин входів. Для порівняння алгоритмів у діапазоні реальних довжин входів, що визначаються областю застосу вання програмної системи, необхідно знати точну кількість операцій, заданих алгоритмом, тобто його функцію працеємності.

Нехай D_A – множина припустимих конкретних проблем задачі, що вирішується алгоритмом A, а його елемент D ϵ D_A є конкретною проблемою (вхід алгоритму A) вимірності n. Множина D є кінцевою впорядкованою множиною з n елементів d_i, що являють собою слова фіксованої довжини в абетці {0,1}: $D = \{d_i , i = 1,\ldots, n,\}$, $|D| = n$.

Працеємністю алгоритму A на вході D називають кількість базових операцій у прийнятій моделі обчислень, що задаються алгоритмом на цьому вході (позначають $f_A(D)$). Значенням функції працеємності для будь-якого припустимого входу D є ціле додатне число [10].

Іноді при детальному аналізі алгоритмів виявляється, що не завжди працеємність алгоритму на одному вході D довжини n, де $n = |D|$, співпадає з його працеємністю на іншому вході такої же довжини. Розглянемо припустимі входи алгоритму довжини n. У загальному випадку існує підмножина, для більшості алгоритмів власна, множини D_A, що включає всі входи вимірності n, яку позначимо D_n:

D_n = {D | $|D|$ = n}. Т.я. елементи d_i є словами фіксованої довжини в абетці {0,1}, множина D_n є скінченою (позначимо її потужність M_{Dn}, $M_{Dn}=|D_n|$). Тоді алгоритм А, одержуючи різні входи D з множина D_n, буде, можливо, задавати у деякому випадку найбільшу, а в деякому – найменшу кількість операцій. Виключення складають алгоритми, для яких працеємність визначається тільки довжиною входу (тоді $f_A(n)$).

Найгіршим випадком є найбільша кількість операцій, що задаються алгоритмом А на всіх входах вимірності n, тобто на всіх входах D ϵ D_n:

$$f_A^{\wedge}(n) = \max_{D \in D_n} \{f_A(D)\}.$$

Найкращим випадком є найменша кількість операцій, що задаються алгоритмом А на всіх входах вимірності n, тобто на всіх входах D ϵ D_n:

$$f_A^{\vee}(n) = \min_{D \in D_n} \{f_A(D)\}.$$

Працеємністю алгоритму А у середньому є середня кількість операцій, що задаються алгоритмом А на всіх входах вимірності n:

$$\overline{f}_A(n) = \sum_{D \in D_n} p(d) f_A(D),$$

де $p(D)$ – частота зустрічей входу D для області застосування алгоритму, що аналізується. Якщо всі входи D ϵ D_n є рівноймовірними, то

$$\overline{f}_A(n) = \frac{1}{M_{D_n}} \sum_{D \in D_n} f_A(D).$$

Часова складність алгоритму – асимптотична оцінка у класах функцій, що визначаються позначеннями O або Θ, функції працеємності алгоритму для найгіршого випадку $f_A^{\wedge}(n) = O(g(n))$ або $f_A^{\wedge}(n) = \Theta(g(n))$, де $g(n)$ – функція, що задає клас O або Θ для $f_A^{\wedge}(n)$ [10].

Маємо $f_A^{\wedge}(n) \le \overline{f}_A(n) \le f_A^{\vee}(n)$.

Ємнісна складність та функція об'єму пам'яті. Стан пам'яті моделі обчислень визначається значеннями, записаними у комірках цієї пам'яті. Механізм реалізації моделі обчислень, виконуючи операції, задані алгоритмом, переводить початковий стан пам'яті моделі обчислень (вхідні дані задачі – вхід алгоритму) у кінцевий стан (знайдене алгоритмом рішення задачі). В ході рішення задачі може бути задіяно деяку додаткову кількість комірок пам'яті.

Під об'ємом пам'яті, що потрібний алгоритму А для входу, заданого множиною D, розуміють максимальну кількість комірок пам'яті моделі

обчислень, задіяних у ході виконання алгоритму. Функція об'єму пам'яті алгоритму для входу D, що позначимо $V_A(D)$, є теж цілим додатним числом [10].

Аналогічно маємо найгірший ($V_A^{\wedge}(D)$), середній ($\overline{V}_A(D)$) та найкращий ($V_A^{\vee}(D)$) випадки.

Ємнісна складність алгоритму – це асимптотична оцінка в класах функцій, що визначаються позначеннями O або Θ, функції об'єму пам'яті алгоритму для найгіршого випадку $V_A^{\wedge}(n) = O(h(n))$ або $V_A^{\wedge}(n) = \Theta(h(n))$, де $h(n)$ – функція, що задає клас O або Θ для $V_A^{\wedge}(n)$.

Маємо $V_A^{\wedge}(n) \le \overline{V}_A(n) \le V_A^{\vee}(n)$ [10].

Ресурсна характеристика і ресурсна складність алгоритму. Поняття ресурсної ефективності алгоритму включає потрібний алгоритму (при вирішенні задач вимірністю n) ресурс процесору та ресурс пам'яті. Тому ресурсною характеристикою алгоритму у найгіршому, середньому або найкращому випадку називають впорядковану пару функцій, а саме відповідні функції працеємності та функція об'єму пам'яті [10]:

$\Re_n^*(A) = \left\langle f_A^*(n), V_A^*(n) \right\rangle$, де $* \in \{\wedge, \vee, -\}$.

Ресурсною складністю алгоритму у найгіршому, середньому або найкращому випадку називають впорядковану пару класів функцій, заданих асимптотично позначеннями O або Θ (як відповідні випадку часова складність і ємнісна складність алгоритму), - $\Re_c^*(A) = \left\langle O(g(n)), O(h(n)) \right\rangle$.

Іноді більш наглядним є перехід в оцінці ємнісної складності від загального об'єму пам'яті моделі обчислень до об'єму додаткової пам'яті, що потрібна алгоритму, якщо об'єми пам'яті для входу й результату однакові для всіх порівнюваних алгоритмів. Тоді позначення всі зберігаються, вони використовують з відповідним уточненням. Наприклад, позначення ресурсної складності для алгоритму сортування вставками, для якого $f_A^{\wedge}(n) = \Theta(n^2)$, а потрібна додаткова пам'ять фіксована й не залежить від довжини масиву, що сортується. Для цього алгоритму $\Re_c^{\wedge}(A) = \left\langle \Theta(n^2), \Theta(1) \right\rangle$.

Функції ресурсної ефективності комп'ютерних алгоритмів та їх програмних реалізацій [10]. Розглянемо компоненти функції ресурсної

ефективності програмної реалізації алгоритму. Нехай D_A – конкретна множина початкових даних алгоритму.

$V_{exe}(D_A)$ – ресурс оперативної пам'яті в області коду, потрібний під розміщення машинного коду, що реалізує даний алгоритм. Ресурс співставляють з об'ємом EXE-файлу, з урахуванням принципів організації керування програмною системою. Як правило, для одної й тієї ж задачі, короткі за обсягом реалізуючого програмного коду алгоритми мають гірші часові оцінки ніж більш довгі, т.я. швидкі алгоритми в більшості випадків є доволі складними. Переважно для невеликих програм об'єм області коду не залежить від D_A. Для великих програмних систем або комплексів модулі керування обчислювальним процесом за визначених початкових даних можуть підгрузити до ОП додаткові фрагменти коду, причому до одержання такої адаптивної структури програмного забезпечення можуть приводити результати порівняльного аналізу ресурсної ефективності алгоритмів;

$V_{ram}(D_A)$ – ресурс додаткової ОП в області даних, що потрібний алгоритму під тимчасові комірки, масиви й структури. Ресурс пам'яті для входу та виходу алгоритму не враховується в цій оцінці, т.я. є незмінним для всіх алгоритмів рішення даної задачі. Часто більш швидкі алгоритми рішення задачі вимагають більшого обсягу додаткової пам'яті.

$V_{st}(D_A)$ – ресурс ОП в області стеку, що потрібний алгоритму для організації виклику внутрішніх процедур та функцій. Величина даного ресурсу суттєво залежить від того, в якому підході (ітераційному, рекурсивному) реалізований алгоритм. Потрібний обсяг пам'яті в області стеку може бути критичним при рекурсивній реалізації по відношенню до вимірності задачі, якщо дерево рекурсії досить «глибоке». Якщо алгоритм реалізують в об'єктно-орієнтованому середовищі програмування, то вимоги до ресурсу стеку можуть бути значними за рахунок довгих ланцюгів викликів методів, пов'язаних із спадкуванням в об'єктах.

$T(D_A)$ – потрібний алгоритму ресурс процесора – оцінка часу виконання даного алгоритму на даному комп'ютері. Оцінка визначається функцією працеємності алгоритму в залежності від характеристичних особливостей множини початкових даних. Перехід від функції працеємності до часової оцінки пов'язаний з визначенням середньозваженого часу $t_{оп}$ виконання узагальненої базової операції в мові реалізації

алгоритму на даному процесорі й комп'ютері. Отримання точної функції часу виконання, яка враховує всі особливості архітектури комп'ютера, представляє собою доволі складну задачу, для оцінки зверху можна використати функцію працеємності для найгіршого випадку за даної вимірності $f_A^{\wedge}(n)$. У переважній кількості випадків може бути використана функцію працеємності для середнього випадку $\overline{f}_A(n)$. Середньозважений час $t_{оп}$ може бути отриманий дослідним шляхом у середовищі реалізації алгоритму.

Функції ресурсної ефективності алгоритму та його програмної реалізації [10]. Один з підходів до побудови ресурсних функцій є вартісний, що використовує уведені вагові коефіцієнти. За ним функція ресурсної ефективності алгоритму для входу D набуває вигляду

$$\Psi_A(D) = C_V V_A(D) + C_f f_A(D),$$

А функція ресурсної ефективності програмної реалізації

$$\Psi_{AR}(D) = C_{exe}V_{exe}(D) + C_{ram}V_{ram}(D) + C_{st}V_{st}(D) + C_t T_A(D),$$

де конкретні значення вагових коефіцієнтів задають питомі вартості ресурсів, визначені умовами застосування алгоритму та специфікою програмної системи.

Вибір раціонального алгоритму A_r може відбутися за заданих значень вагових коефіцієнтів за критерієм мінімуму функції $\Psi_{AR}(D)$, обчисленої для всіх алгоритмів-претендентів з множини A. Нехай множина алгоритмів-претендентів складається з m елементів, A={A_i| i =1,…,m}, тоді $A_r(D): \Psi_{ARr}(D) = \min\limits_{A_i}\{\Psi_{ARi}(D)\}$. Формально, виконавши такі обчислення для всіх входів D з множини D_A можна визначити ті множини входів, за яких один з алгоритмів-претендентів буде найбільш раціональним, але підхід непрактичний внаслідок дуже великої кількості елементів у множині D. Практичний метод пов'язаний з використанням оцінки в середньому або найгіршому випадку – $\Psi_A^*(n) = C_V V_A^*(n) + C_f f_A^*(n)$, де $* \in \{\wedge, \vee, -\}$ позначення найгіршого, найкращого та середнього випадку ресурсної функції, а функція ресурсної ефективності програмної реалізації: $\Psi_{AR}(n) = C_{exe}V_{exe}(n) + C_{ram}V_{ram}(n) + C_{st}V_{st}(n) + C_t T_A(n)$.

Для вибору раціонального алгоритму визначимо функцію R(n), значенням якої є номер алгоритму r, для якого $R(n) = r$, $r \in \{1,...,m\}$;

$r = \arg\min_{i=1,m}\{\Psi_{AR_i}(n)\}$. Тоді на досліджуваному інтервалі вимірностей входу $[n_1, n_2]$ можливі такі випадки:

- функція $R(n)$ приймає однакові значення на $[n_1, n_2]$, тобто $R(n_{i,}) = R(n_j) = k, \forall n_{i,} n_j \in [n_1, n_2]$, алгоритм з номером k є раціональним за вказаною ресурсною функцією на всьому інтервалі $[n_1, n_2]$;
- функція R(n) приймає різні значення на $[n_1, n_2]$, тобто $\exists n_{i,} n_j \in [n_1, n_2]: R(n_{i,}) \neq R(n_j)$, тому декілька алгоритмів є раціональними за вказаною ресурсною функцією для різних вимірностей входу на інтервалі $[n_1, n_2]$ (можна виділити підінтевали, на яких $R(n_i) = const$ та реалізувати адаптивний за вимірністю входу вибір раціонального алгоритму.

Отже, на основі функції ресурсної ефективності можна вибрати алгоритм, що має мінімальну ресурсну вартість за даних ваг на деякому підінтервалі досліджуваного інтервалу вимірностей. На основі функцій ресурсної ефективності можна більш детально дослідити задачі вибору раціонального алгоритму з точки зору характеристик конкретного комп'ютера та зовнішніх вимог до алгоритму. Такий підхід приводить до побудови чотирьохвимірного простору аргументів, координатами яко го є значення ресурсних функцій (об'єм пам'яті коду, додаткових даних, стеку та часова оцінка). В такому просторі зовнішні вимоги до алгоритму (його програмної реалізації), задані у вигляді обмежуючих сегментів, задають чотирьохвимірну область W припустимих значень. Досліджуваній програмній реалізації алгоритму на даному комп'ютері для кожного значення вимірності задачі n в цьому просторі відповідає точка $X(n)$ з координатами $X(n) = (V_{exe}(n), V_{ram}(n), V_{st}(n), T_A(n))$ де координати обчислюються за середнім або найгіршим випадком для даної вимірності.

За такого підходу є можливість дослідження алгоритмів на «стійкість» по відношенню до характеристик множини початкових даних (насамперед вимірності задачі) у заданій області вимог. Крім того, можливе дослідження області області W та характеристик комп'ютерів (опосередковано через $T(n)$) з метою вибору раціонального комп'ютеру для даного алгоритму рішення задачі з урахуванням зовнішніх вимог.

Способи побудови комплексних критеріїв оцінки якості алгоритмів [10]. Функція ресурсної ефективності обчислювальних алгоритмів як

функція вимірності входу враховує ресурси комп'ютера, що потрібні алгоритму. Це базова ресурсна функція, компоненти якої можуть бути модифіковані з урахуванням додаткових вимог до характеристик алгоритму з боку програмної системи. Така модифікація може бути виконана уведенням додаткових компонент, що враховують спеціальні вимоги до алгоритмічного забезпечення, або уведенням функціональних залежностей для вартісних коефіцієнтів з аргументом вимірності входу в компонентах функції ресурсної ефективності.

Розглянемо деякі варіанти побудови додаткових компонентів.

Врахування вимог точності. Такі вимоги звичайно виникають в задачах оптимізації, що алгоритмами, які базуються на використанні чисельних методів, а також при отриманні рішень для NP-повних задач, коли точний розв'язок не може бути отриманий за поліноміальний час й використовують спеціальні алгоритми для одержання ε-поліноміальних наближень для NP-повних задач. Врахування потрібної точності одержуваного рішення можна зробити для наведеного вартісного підходу такими способами.

1. Базується на безпосередньому врахуванні точності в компонентах. Точність для оптимізаційних алгоритмів, що розуміється як точність, що забезпечується процесом ітераційної збіжності, задається вхідним значенням ε, яке сильно впливає на працеємність. При цьому функція працеємності (а тому й час виконання у середньому T(n)) залежить від вимірності та точності – T(n, ε). Можливо й додаткові витрати пам'яті теж залежать від точності, тому функція ресурсної ефективності програмної реалізації є функцією вимірності та точності $\Psi_{AR} = \Psi_{AR}(n, \varepsilon)$.
2. Використовують побудову базової функції за фіксованого мінімально прийнятного значення точності ε = ε_0. Ресурсні витрати на подальше підвищення точності можуть бути внесені до окремого компоненту функції ресурсної ефективності з власною вартісною вагою:

$$E(n, \frac{\varepsilon_0}{\varepsilon}) = T_\varepsilon(n, \frac{\varepsilon_0}{\varepsilon}) + V_{\varepsilon,ram}(n, \frac{\varepsilon_0}{\varepsilon}), \quad \Psi_{AR}(n, \varepsilon) = \Psi_{AR}(n, \varepsilon_0) + E(n, \frac{\varepsilon_0}{\varepsilon}), \varepsilon < \varepsilon_0,$$

де компонент T_ε враховує у відносних одиницях зміну часової ефективності при зміні точності, компонент $V_{\varepsilon,ram}$ – зміну додатково потрібної пам'яті, $\Psi_{AR}(n, \varepsilon_0)$ – значення базової функції при ε = ε_0.

Врахування вимог за часовою стійкістю. Вимоги, пов'язані з часовими особливостями функціонування апаратних засобів (наприклад, бортових обчислювальних комплексів) обумовлені тим, що програмне забезпечення таких систем має задовольняти жорстким часовим обмеженням та мати властивість часової стійкості за даними (коли різні входи обчислювального алгоритму за фіксованої довжини входу дають невеликі зміни часу виконання). Цей показник забезпечує стійкість розрахункового часу відгуку системи на зовнішні впливи. Врахування вимоги часової стійкості можна виконати за допомогою використання поняття інформаційної чуттєвості алгоритму, що відображає зміну значень функції працеємності для різних входів фіксованої довжини. Кількісна міра інформаційної чуттєвості визначається за дослідження алгоритму методами математичної статистики. Додатковий компонент функції ресурсної ефективності в цьому випадку одержують як добуток відповідного вагового коефіцієнту на кількісну міру інформаційної чуттєвості алгоритму.

Функціональні вартісні коефіцієнти. Деякі спеціальні вимоги до програмного забезпечення можна врахувати у функції ресурсної ефективності шляхом зміни значень вартісних коефіцієнтів при зміні вимірності задачі. Це є ефективним за необхідності врахування «порогових» вимог (наприклад, за обсягом потрібної додаткової пам'яті або часу виконання). Т.я. функція ресурсної ефективності програмної реалізації алгоритму є функцією вимірності входу, а порогові вимоги надаються у вимірностях компонент функції, то розглядають вартісні коефіцієнти як функції значень компонентів, наприклад, $C_{ram} = C_{ram}(V_{ram}(n))$ або за наявності порогових значень аргументу функції із сходинками (Хевісайда): $C_{ram}(n) = \sum_{j=1}^{k} C_{ram\,j} H(V_{ram}(n) - V_{ram\,j-1})$.

Лекція 13. Прогнозування часу виконання програмних реалізацій за функцією працеємності. Спеціальні класифікації обчислювальних алгоритмів

Результати порівняльного аналізу алгоритмів за часом виконання їх програмних реалізацій не завжди співпадають з результатами аналізу

за функцією працеємності внаслідок різної частоти зустрічі операцій та різниці у часі їх виконання, що пов'язані з особливостями компілятора, операційної системи та архітектури. Просте експериментальне рішення задачі побудови часових характеристик – дослід з програмною реалізацією та подальша екстраполяція отриманих значень. Такий підхід не є достатньо точним з причин відсутності інформації про властивості алгоритму, що реалізується, й приводить до необхідності врахування функції працеємності при вирішенні задачі прогнозування. З метою одержання більш достовірних результатів потрібний перехід від функції працеємності до функції часу виконання програмної реалізації алгоритму. При цьому розглядають середні та/або найгірші випадки для фіксованої вимірності задачі: $T_A(n) = T_A(f_A(n), r_1, r_2, \ldots, r_k)$ (де $T_A(n)$ – функція часової ефективності, що задає час виконання алгоритму у середньому або найгіршому випадку, $f_A(n)$ – функція працеємності алгоритму у середньому або найгіршому випадку, $r_1, r_2, \ldots, r_k$ ($k \le n$) – параметри, що враховують особливості середовища реалізації. При побудові функції $T_A(n)$ стикаємося з множиною факторів, що пов'язані з особливостями середовища реалізації (мова програмування, компілятор, операційна система, комп'ютер), які породжують суттєві труднощі. Але все одно різноманітні методи побудови часових оцінок є.

Метод типових задач [10] враховує лише тип задачі, що вирішується. Ідея методу: в межах фіксованого типу задач, наприклад, задач, пов'язаних з науково-технічними розрахунками, середній час на 1 узагальнену операцію є стійким внаслідок використання однакових типів даних та близької частоти зустрічі операцій. Метод передбачає проведення сукупного аналізу алгоритму по працеємності та перехід до часової оцінки його програмної реалізації на основі належності задачі до одного з наступних типів: задачі науково-технічного характеру з перевагою операцій з операндами дійсного типу; задачі дискретної математики з перевагою операцій з операндами цілого типу; задачі баз даних з перевагою операцій з операндами рядкового типу.

Далі на основі аналізу множини реальних програм для рішення відповідних типів задач визначається частота зустрічі операцій. Одержані результати покладені в основу створення відповідних тестових програм, що породжують задану частоту зустрічі операцій для заданих типів даних. На основі дослідів з цими тестовими програмами й визначається середній час на узагальнену операцію в даному типі задач –

$\bar{t}_{\text{тип задачі}}$, далі оцінюється загальний час роботи програмної реалізації алгоритму для даного комп'ютеру й даного середовища реалізації: $T_A(n) = \bar{f}_A(n)\bar{t}_{\text{тип задачі}}$.

Перевагою методу є те, що значення $\bar{t}_{\text{тип задачі}}$ визначається одноразово для даної мови, компілятора, операційної системи, комп'ютера, а потім використовується для одержання часових оцінок програмних реалізацій алгоритмів на основі належності задачі до одного з указаних типів. Такі оцінки не мають великої точності, але можуть бути досить просто отримані для будь-якої програмної реалізації, виконуваної на даному комп'ютері.

Метод пооперaційного аналізу [10]. В методі представляють функцію працеємності алгоритму як суму працеємностей за базовими операціями та визначають середній час виконання кожної з них у вибраному середовищі реалізації. Метод включає етапи: отримання пооперaційних функцій працеємності $\bar{f}_{A\text{оп}i}(n)$ для кожної з використаних алгоритмом базових операцій з урахуванням типів даних (одержання таких пооперaційних функцій вимагає деяких витрат, але іноді дозволяє отримати вагомі результати щодо вибору раціональних алгоритмів); дослідне визначення середнього часу виконання даної базової операції $\bar{t}_{\text{оп A}i}$ на конкретній обчислювальній машині в середовищі обраної мови програмування й операційної системи за допомогою спеціальної тестової програми; обчислення очікуваного часу виконання програмної реалізації як суми добутків пооперaційної працеємності на середні часи операцій $T_A(n) = \sum_{i=1}^{k} \bar{t}_{\text{оп A}i}\bar{f}_{A\text{оп}i}(n)$, де k – кількість базових операцій; $\bar{f}_{A\text{оп}i}(n)$ – функція працеємності алгоритму по операції i, i =1, ,…, k, $\bar{t}_{\text{оп A}i}$ – експериментальний час виконання операції i.

Похибка прогнозу в даному методі виникає за рахунок того, що реальний потік операцій, що породжуються алгоритмом, не завжди співпадає з потоком у дослідній програмі, що визначає середній час виконання базової операції. Вказане може відбуватися внаслідок деяких особливостей архітектури процесора, внутрішніх алгоритмів керування кеш-пам'яттю тощо. Але часто тільки методом пооперaційного аналізу можна виявити особливості раціонального застосування конкретного алгоритму вирішення задачі.

Емпіричний метод одержання часових оцінок на основі функції працеємності [10]. В даному методі:

1. Виконується загальний аналіз працеємності алгоритму без поділу на операції (визначають функцію працеємності алгоритму для середнього випадку $\overline{f}_A(n)$ в узагальнених базових операціях прийнятої моделі обчислень).
2. Проводяться обчислювальні експерименти з програмною реалізацією алгоритму (для кожного з обраних значень вимірності n_i, $i = 1,\ldots, m$, виконують деяку кількість експериментів n_e, $j = 1,\ldots, n$, з різними початковими даними, в ході яких визначаються часи виконання $t_{ej}(n_i)$, на основі яких розраховують середній експериментальний час виконання:
$$\bar{t}_e(n_i) = \left(\frac{1}{n_e}\right)\sum_{j=1}^{n_e} \bar{t}_{e\,j}(n_i)\,.$$
3. Розраховують час середній ($\bar{t}_{\text{оп A}}(n_i)$) та за всім експериментом ($\bar{t}_{\text{оп A}e}$), а саме за відомою функцією працеємності алгоритму для середнього випадку $\overline{f}_A(n)$ обчислюють середній час на узагальнену базову операцію, породжуваний даним алгоритмом, компілятором, ОС та комп'ютером для даної вимірності:
$$\bar{t}_{\text{оп}A}(n_i) = \left(\frac{\bar{t}_e(n_i)}{\bar{f}_A(n_i)}\right),\quad \bar{t}_{\text{оп}Ae}(n_i) = \left(\frac{1}{m}\right)\sum_{i=1}^{m_e} \bar{t}_{\text{оп}A}(n_i),$$
де m – кількість значень вимірності задачі, що тестується.
4. Виконують прогноз часової ефективності за умови припущення про стійкість середнього часу виконання узагальненої базової операції:
$$\overline{T}_A(n) = \bar{t}_{\text{оп}Ae}(n_i)\,\bar{f}_A(n),\ \ n \notin n_i,\ \ i = \overline{1,m}.$$

Наприклад, маємо наступні дослідні дані за алгоритмом сортування методом прямого включення (реалізація: конкретна мова, конкретна ОС, конкретний комп'ютер, конкретний процесор) [10]. Тестові варіанти отримані стандартним генератором псевдовипадкових чисел з рівномірним розподілом. Для кожного фіксованого значення вимірності масиву виконано $n_e = 10^6$ дослідів, їх результати надані у таблиці 13.1 (t_{sum} – час у секундах за всіма дослідами; $\bar{t}_e(n_i)$ – середній час у секундах на сортування одного масиву; $\bar{t}_{\text{оп A}}(n_i)$ – середній час на узагальнену базову операцію в наносекундах).

Таблиця 13.1. Дослідні дані для визначення оцінок емпіричним методом

i	n_i	$\overline{f}_A(n)$	t_{sum}	$\bar{t}_{e}(n_i)$	$\bar{t}_{\text{оп A}}(n_i)$
1	50	6815	17.51	0.00001751	2.5705
2	60	9680	24.45	0.00002445	2.5266
3	70	13045	32.52	0.00003252	2.4935
4	80	16910	41.80	0.00004180	2.4723
5	90	21275	52.30	0.00005230	2.4586
6	100	26140	63.88	0.00006388	2.4440
7	200	102290	244.90	0.00024490	2.3942
				$\bar{t}_{\text{опA}e}$	2.4800

Прогнозовані результати для n =300:

$$\overline{f}_A(n) = 228440,\ \mathrm{T_A}(n) = \bar{t}_{\text{оп A}}\,\overline{f}_A(n) = 0.00056652.$$

Прогнозовані результати для n =400:

$$\overline{f}_A(n) = 404590,\ \mathrm{T_A}(n) = \bar{t}_{\text{оп A}}\,\overline{f}_A(n) = 0.00100337.$$

Точність прогнозу може бути підвищена за умови отримання залежності $\bar{t}_{\text{оп A}}(n_i)$ від розмірності задачі.

Аналіз впливу вимірності задачі на середній час виконання узагальненої базової операції. Аналізуючи функціональну залежність $\bar{t}_{\text{оп A}}(n)$ від n [10] (враховуючи, що для великих значень n середній час прямує до сталої) можна в якості пояснення для спостережуваної в досліді залежності $\bar{t}_{\text{оп A}}(n)$ від вимірності n мати наступне: фрагмент, що обумовлює в оцінці для середнього випадку $\overline{f}_A(n)$ головний порядок функції працеємності, має дещо іншу частоту зустрічі операцій, ніж фрагменти, що породжують асимптотично слабкіші порядки в $\overline{f}_A(n)$, а це приводить до того, що в області малих значень вимірностей загальний середній час на операцію буде відрізнятися від середнього часу в області великих значень вимірностей. Функція працеємності має адитивний характер і її можна представити у вигляді:

$$\overline{f}_A(n) = \sum_{i=0}^{k} \overline{f}_i(n),\ \ \overline{f}_0(n) = const,$$

де k – кількість функціональних адитивних компонент функції працеємності, компоненти суми $\overline{f}_i(n)$ можуть бути впорядковані за асимптотичною ієрархією ($\overline{f}_i(n) \succ \overline{f}_{i-1}(n), \forall i = \overline{1,k},$ причому $\Theta\left(\overline{f}_A(n)\right) = \Theta\left(\overline{f}_k(n)\right)$).

Поведінка $\overline{f}_A(n)$ в області малих значень вимірностей визначається як головним членом, так і більш слабкими асимптотичними компонентами нижчого порядку, що приводить до необхідності врахування їхнього впливу на значення функції $\overline{f}_A(n)$, а тому й часові оцінки.

За умови, що в межах області застосування алгоритму ($n_{\min} \le n \le n_{\max}$) виконується $\overline{f}_{k-2}(n_{min}) << \overline{f}_{k-1}(n_{\min})$, при практичному аналізі можна обмежитися розглядом двох основних компонент функції працеємності $\overline{f}_A(n)$ (а саме: $\overline{f}_k(n)$, $\overline{f}_{k-1}(n)$). Нехай $\bar{t}_i$ – середній час узагальненої базової операції для *i*-ої компоненти функції працеємності, тоді середній час узагальненої базової операції алгоритму матиме вигляд:

$$\bar{t}_{\text{оп A}}(n) = \sum_{i=0}^{k} \bar{t}_i \overline{f}_i(n) / \sum_{i=0}^{k} \overline{f}_i(n), \quad \overline{f}_A(n) = \sum_{i=0}^{k} \overline{f}_i(n).$$

Тому одержимо середньозважену оцінку, де компоненти $\bar{t}_i$ мають ваги, що дорівнюють доданкам у функції працеємності (за фіксованого *n*). Розгляд тільки двох старших компонент приведе до виразу:

$$\bar{t}_{\text{оп A}}(n) \approx \frac{\bar{t}_k \overline{f}_k(n) + \bar{t}_{k-1} \overline{f}_{k-1}(n)}{\overline{f}_k(n) + \overline{f}_{k-1}(n)},$$

Звідки

$$\bar{t}_{\text{оп A}}(n) \approx \bar{t}_k \left(\frac{\overline{f}_k(n)}{\overline{f}_k(n) + \overline{f}_{k-1}(n)} + \frac{\bar{t}_{k-1}}{\bar{t}_k} \frac{\overline{f}_{k-1}(n)}{\overline{f}_k(n) + \overline{f}_{k-1}(n)} \right).$$

Впорядкованість за асимптотичною ієрархією компонент функції працеємності та одержаний вираз пояснюють залежність середнього часу узагальненої базової операції від вимірності задачі. Більш того, $\lim_{n\to\infty} \bar{t}_{\text{оп}A}(n) = \bar{t}_k$, а тому в області великих значень вимірності $\bar{t}_{\text{оп A}}(n)$ слабо залежить від значення *n*.

Конкретний вигляд залежності $\bar{t}_{\text{оп A}}(n)$ від вимірності задачі визначається конкретними компонентами функції працеємності алгоритму. Припустимо, що функція працеємності відома (хоча б з точністю до двох асимптотично головних доданків).

Розглянемо найпростіший приклад, функція працеємності алгоритму має вигляд: $\overline{f}_A(n) = a_2 n^2 + a_1 n + a_0$, $k = 2$, в межах області застосування алгоритму значення вимірності таке, шо $a_1 n_{\min} >> a_0$, а тому вплив a_0 можна не враховувати.

Напишемо оцінку часової ефективності програмної реалізації $\bar{t}_A(n) = \bar{t}_2 a_2 n^2 + \bar{t}_1 a_1 n$ й уведемо наступні позначення: $\alpha = \bar{t}_1 / \bar{t}_2$ ($\bar{t}_1 = \alpha \bar{t}_2$), $\beta = a_1 / a_2$ ($a_1 = \beta a_2$). Тоді:

$$\bar{t}_A(n) = \bar{t}_2 a_2 n^2 + \bar{t}_2 \alpha\beta a_2 n = \bar{t}_2 a_2 n(n + \alpha\beta), \quad \bar{f}_A(n) = a_2 n^2 + \beta a_2 n = a_2 n(n + \beta).$$

Функція середнього часу на узагальнену базову операцію буде:

$$\bar{t}_{\text{оп A}}(n) = \bar{t}_2 \frac{n + \alpha\beta}{n + \beta}. \qquad (13.1)$$

А звідси й $\lim\limits_{n\to\infty} \bar{t}_{\text{оп}A}(n) = \bar{t}_2$. Для дослідження поведінки $\bar{t}_{\text{оп A}}(n)$ виконаємо перетворення:

$$\bar{t}_{\text{оп A}}(n) = \bar{t}_2 \frac{n + \alpha\beta + \beta - \beta}{n + \beta} = \bar{t}_2 \left(1 + \frac{\beta(\alpha - 1)}{n + \beta}\right) \qquad (13.2)$$

Тоді, якщо α > 1 (середній час на операцію в лінійній компоненті більший ніж у квадратичній), то $\bar{t}_{\text{оп A}}(n)$ буде зменшуватися із зростанням n, асимптотично наближаючись до $\bar{t}_2$ зверху. Якщо ж α < 1 (середній час на операцію в лінійній компоненті менший ніж у квадратичній), то $\bar{t}_{\text{оп A}}(n)$ буде зростати із зростанням n, асимптотично наближаючись до $\bar{t}_2$ знизу (рис. 13.1).

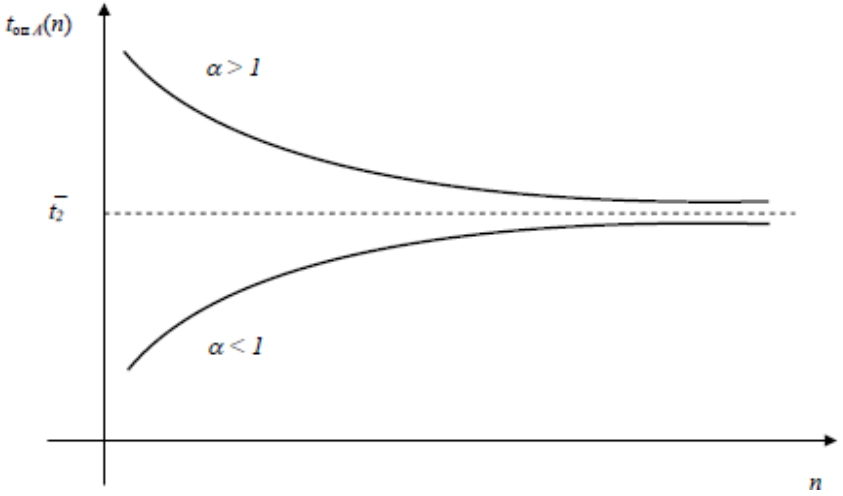


Рис. 13.1. Залежність $\bar{t}_{\text{оп A}}(n)$ від параметру α [10]

Для побудови на практиці прогнозу часової ефективності алгоритму потрібно визначити величини t_2 и α на основі дослідних даних. Значення β буде відоме з функції працеємності алгоритму, одержаної в результаті аналізу, а дослідний ряд значень вимірностей та середніх значень часу (n_i та $\bar{t}_{\text{оп A}}(n_i)$) одержані при визначенні середнього часу виконання узагальненої базової операції.

Розглянемо два *варіанти лінеаризації функції* $\bar{t}_{\text{оп A}}(n)$.

1. Напишемо вираз (1) для функції $\bar{t}_{\text{оп A}}(n)$ у вигляді

$$\bar{t}_{\text{оп A}}(n) = \bar{t}_2 + \bar{t}_2\beta(\alpha - 1)\frac{1}{n+\beta}.$$

Уведемо наступну заміну змінних (лінійно-гіперболічну):

$$y = \bar{t}_{\text{оп A}}(n), \quad x = \frac{1}{n+\beta}, \quad b = \bar{t}_2, \quad a = \bar{t}_2\beta(\alpha - 1).$$

Так як значення β відоме з функції працеємності алгоритму, то значення x можуть бути обчислені апріорно для дослідних значень вимірності задачі n_i. За нових змінних: $y = ax + b$. Методом найменших квадратів можемо побудувати лінійну регресію за дослідних рядом значень та отримати оптимальні значення a^*, b^*. Далі беремо $\bar{t}_2 = b^*$, а α обчислюємо як $\alpha = 1 + \dfrac{a^*}{b^*\beta}$.

2. Припустимо, що область реальних значень вимірностей така, що значення $\dfrac{1}{n^2}$ є доволі малими. Прологарифмуємо вираз (13.1) для функції $\bar{t}_{\text{оп A}}(n)$:

$$\ln(\bar{t}_{\text{оп A}}(n)) = \ln(\bar{t}_2) + \ln(n + \alpha\beta) - \ln(n + \beta).$$

Використаємо наступні допоміжні перетворення:

$$n + \alpha\beta = n\left(1 + \frac{\alpha\beta}{n}\right), \quad n + \beta = n\left(1 + \frac{\beta}{n}\right).$$

Тоді

$$\ln(\bar{t}_{\text{оп A}}(n)) = \ln(\bar{t}_2) + \ln(n) + \ln\left(1 + \frac{\alpha\beta}{n}\right) - \ln(n) - \ln\left(1 + \frac{\beta}{n}\right).$$

Так як за припущенням область реальних значень вимірностей така, що значення $\frac{1}{n^2}$ є доволі малими, то застосовуючи асимптотичне представлення для виразу $\ln(1+z)$ за $z \to 0$ ($\ln(1+z) \approx z + O(z^2)$) одержимо:

$$\ln(\bar{t}_{\text{оп A}}(n)) = \ln(\bar{t}_2) + \frac{\alpha\beta}{n} - \frac{\beta}{n} = \ln(\bar{t}_2) + \frac{\beta}{n}(\alpha - 1).$$

Уведемо заміну (логарифмічно-гіперболічну):

$$y = \ln(\bar{t}_{\text{оп A}}(n)), \quad x = \frac{\beta}{n}, \quad b = \ln(\bar{t}_2), \quad a = \alpha - 1.$$

За нових змінних, як і в попередньому варіанті, $y = ax + b$. Методом найменших квадратів можемо побудувати лінійну регресію за дослідних рядом значень та отримати оптимальні значення a^*, b^*, з яких $\bar{t}_2 = e^{b^*}$, $\alpha = 1 + a^*$.

Наведений підхід узагальнюється для випадків, якщо адитивна функція працеємності є поліномом *k*-ої степені з невідомими коефіцієнтами, та, якщо деякі адитивні компоненти функції працеємності мають логарифмічні або полілогарифмічні множники.

Наприклад, застосуємо наведений метод для одержання прогнозу часу виконання програмної реалізації алгоритму сортування вставками. Нехай маємо результати аналізу дослідних даних для визначення коефіцієнтів функції $\bar{t}_{\text{оп A}}(n)$, вигляд якої визначається за формулой (13.1) для обох наведених варіантів лінеаризації (a = 13,164342, b = 2,321930 та a = 1,034124, b = 0,847156, відповідно). В таблиці 13.2 наведені результати дослідів та прогнози часу виконання для програмної реалізації алгоритму сортування вставками, отримані на основі функції працеємності та обчислених коефіцієнтів залежності середнього часу узагальненої операції від розмірності задачі.

Наведені дані показують підвищення точності прогнозу у порівнянні з використанням середнього часу узагальненої базової операції (врахування залежності середнього часу на узагальнену базову операцію від розмірності задачі дозволило більш ніж на порядок зменшити похибку прогнозу часу виконання).

Таблиця 13.2. Прогноз часу виконання програмної реалізації алгоритму сортування вставками [10]

i	n_i	$\bar{f}_A(n)$	t_{sum}	$\bar{t}_e(n_i)$	$\bar{t}_{\text{оп A}}(n_i)$
Дослідні результати					
1	50	6812	17.51	0.00001751	2.5705
2	60	9677	24.45	0.00002445	2.5266
3	70	13042	32.52	0.00003252	2.4935
4	80	16907	41.80	0.00004180	2.4723
5	90	21272	52.30	0.00005230	2.4586
6	100	26137	63.88	0.00006388	2.4440
7	200	102287	244.90	0.00024490	2.3942
Прогнозовані результати – 1 вар. лінеарізації β = 4.60 t_2 = 2.333001 α = 2.232515					
			Прогноз	Дослід	Похибка %
		300	228437	0.00054330	0.554
		400	404587	0.00095690	0.451
Прогнозовані результати – 2 вар. лінеарізації β = 4.60 t_2 = 2.333001 α = 2.034124					
			Прогноз	Дослід	Похибка %
		300	228437	0.00054127	0.374
		400	404587	0.00095500	0.199

Спеціальні класифікації обчислювальних алгоритмів. Вказують на характерні властивості алгоритмів, важливі для їх дослідження та вирішення питання вибору алгоритму, раціонального в даних умовах застосування. Це класифікації алгоритмів, що відображають складнісні оцінки алгоритмів та вплив на ці оцінки характеристичних особливостей множини початкових даних, породжених задачею. Традиційна класифікація пов'язана з теорією складності обчислень, але об'єктами класичної теорії алгоритмів є обчислювальні задачі й прийняті в такій класифікації класи задач не можна автоматично перенести на алгоритми рішення цих задач. Це пов'язано з тим, що теорія складності оперує класами задач, а не класами алгоритмів, а визначення класів задач, за виключенням класів P й EXP, не включають явно складнісних оцінок. Щодо інших спеціальних класифікацій, що враховують деякі вимоги до програмних засобів та систем, то, наприклад, вимога часової стій

кості може бути врахована в класифікації алгоритмів за інформаційною чуттєвістю.

Класи відкритих та закритих задач, теоретична нижня границя часової складності [10]. Це класифікація задач, основою якої є порівняння доведеної нижньої границі часової складності задачі та оцінки найбільш ефективного з існуючих алгоритмів її рішення. На основі такої класифікації можна стверджувати про теоретичну можливість покращення часової ефективності алгоритму. На сьогодні існують доведення нижніх границь часової складності для множини задач.

Теоретична нижня границя складності. Нехай маємо обчислювальну задачу Z, а D_Z множина конкретних припустимих проблем даної задачі. Нехай: R_Z – множина вірних рішень, Ver – верифікаційна функція задачі, D ϵ D_Z – конкретна припустима проблема задачі, A_Z – повна множина всіх алгоритмів (що включає відомі та майбутні алгоритми), які вирішують задачу Z , тобто будь-який алгоритм A з A_Z застосовуваний до будь-якої D ϵ D_Z, закінчується та дає вірну відповідь ($f_A(D) \neq \infty$). У загальному випадку існує підмножина D_{nz} (для більшості задач власна) множини D_Z, що включає всі конкретні проблеми вимірності n: $D_{nz} = \{D | D \epsilon D_Z, |D| = n\}$.

Для порівняння двох функцій за асимптотикою їх зростання використаємо відношення: $f(n) \prec g(n) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

Вважатимемо, що маємо теоретично доведену нижню границю часової складності задачі $f_{th\ lim}(n)$ як Θ оцінку деякої функції, аргументом якої є довжина входу, якщо теоретично неможливо запропонувати алгоритм, що розв'язує дану задачу асимптотично швидше, ніж з оцінкою $f_{th\ lim}(n)$. Тобто є строге доведення, що будь-який алгоритм рішення даної задачі має у найгіршому випадку на вході довжини n часову складність не краще ніж $f_{th\ lim}(n)$ і дана функція є точною верхньою границею множини функцій, утворючих асимптотичну ієрархію, для яких таке доведення можливе. Інакше кажучи, для довільної функції $g(n) \succ f_{th\ lim}(n)$ вказана властивість не може бути доведеною.

Визначимо множину F_{lim}, що складається з функцій f_{lim}:

$$F_{\lim} = \{ f_{\lim} \mid \forall A \in A_Z, \forall n > 0, f_A^{\wedge}(n) = \Omega(f_{\lim}(n)) \},$$

де $f_A^{\wedge}(n)$ – функція працеємності для найгіршого випадку за всіма входами вимірності n (для будь-якого D ϵ D_{nz}). Тоді $f_{th\ lim}(n) = \sup_{\prec}\{F_{\lim}\}$.

Прикладом задачі з теоретично доведеною нижньою границею складності є задача сортування масиву порівняннями. Для неї доведено неможливість відсортувати масив елементів швидше ніж за $\Theta(\log_2(n!))$ ($f_{thlim}(n) = \Theta(\log_2(n!) = \Theta(n\log_2(n) - n\log_2(e) + 0.5\log_2(2\pi n)) = \Theta(n(\log_2(n) - \log_2(e)) + 0.5\log_2(n) + 0.5\log_2(2\pi)))$.

Тривіальна нижня границя складності. Незалежно від того, доведена або ні теоретична нижня границя часової складності, можна для великої кількості обчислювальних задач вказати нижню границю у вигляді Θ-оцінки деякої функції на основі раціональних міркувань. Такого типу оцінку звичайно будують на основі припущень: оцінка $\Theta(n)$, якщо для розв'язання задачі необхідно щонайменше обробити всі початкові дані (пошук максимуму в масиві, множення векторів тощо); оцінка $\Theta(1)$ для задач з не обов'язковою обробкою всіх початкових даних (пошук у списку за ключем, за умови, що сам список є входом алгоритму). Позначимо таку оцінку через $f_{tr}(n)$ та припустимо, що вона існує для задачі Z, тоді можна вказати такі можливі співвідношення між оцінками $f_{th\ lim}(n)$ та $f_{tr}(n)$: оцінка $f_{th\ lim}(n)$ не доведена для задачі Z й для даної задачі приймають оцінку $f_{tr}(n)$; оцінку $f_{th\ lim}(n)$ доведено, тоді $f_{tr}(n) \prec f_{th\ lim}(n)$ або $f_{tr}(n) = f_{th\ lim}(n)$.

Оцінка складності найкращого відомого алгоритму. Розглянемо множину A_R всіх відомих алгоритмів, що розв'язують задачу, очевидно, що $A_R \subset A_Z$, множина A_R скінчена. Визначимо асимптотично кращий за складністю алгоритм, позначимо його A_{min} (на практиці часто алгоритми з найкращими асимптотичними оцінками мають погані за працеємністю результати на малих вимірностях внаслідок великих коефіцієнтів при головному порядку функції працеємності). Якщо існує декілька алгоритмів з мінімальною за всією множиною A_R асимптотичною оцінкою, то вибирають один з них як алгоритм-представник. Формально визначимо множину алгоритмів $A_M = \{A_m\}$ як підмножину відомих алгоритмів A_R, які мають мінімальну асимптотичну оцінку працеємності, та виділимо у множині алгоритмів A_M будь-який алгоритм: $A_M = \{A_m | \neg\exists A\ \epsilon\ A_R: f_A(n) \prec f_{Am}(n)\}$, $|A_M| \geq 1$, $A_{min}\ \epsilon\ A_M$. Позначимо працеємність цього алгоритму $f_{Amin}(n)$.

Класифікація задач за співвідношенням теоретичної нижньої границі часової складності задачі та складності найбільш ефективного з існуючих алгоритмів. Порівнюючи отримані оцінки для деякої задачі $f_{th\ lim}(n)$,

якщо існує, $f_{tr}(n)$ та $f_{Amin}(n)$ можна використати наступну класифікацію задач:

Клас задач THCL(Theoretical close). Це задачі, для яких $f_{th\ lim}(n)$ існує та $f_{Amin}(n) = \Theta(f_{th\ lim}(n))$. Це клас теоретично замкнених задач (за часовою складністю), оскільки існує алгоритм, що розв'язує дану задачу з асимптотичною часовою складністю рівною теоретично доведеній нижній границі. Розробка нових або модифікація відомих алгоритмів з цього класу може тільки бути спрямованою на покращення коефіцієнту при головному порядку в функції працеємності, $f_A(n) = \Omega(f_{th\ lim}(n))$ для будь-якого A ϵ A_Z та $f_{Amin}(n) = \Theta(f_{th\ lim}(n))$.

Наприклад, для задачі пошуку максимуму в масиві $f_{th\ lim}(n) = \Theta(n)$, $f_{Amin}(n) = \Theta(n)$; для задачі сортування масиву з використанням порівнянь $f_{th\ lim}(n) = \Theta(n \log_2 n)$, $f_{Amin}(n) = \Theta(n \log_2 n)$.

Класс задач PROP(Practical open). Це задачі, для яких $f_{th\ lim}(n)$ існує та $f_{Amin}(n) \succ f_{th\ lim}(n)$. Для задач з цього класу існує відстань між теоретичною нижньою границею складності та оцінкою найбільш ефективного з існуючих алгоритмів її розв'язання. Для задач класу є практичною проблема розробки алгоритму, що має доведену теоретичну нижню границю часової складності. Якщо такий алгоритм буде розроблений, задача буде перенесена до класу THCL. Іноді на основі самого доведення теоретичної нижньої границі часової складності може бути побудований алгоритм рішення даної задачі, що визначає невелику кількість задач в даному класі. Доведення можуть спиратися на ресурсні вимоги (за об'ємом додаткової пам'яті тощо).

Клас задач THOP (Theoretical open). Це задачі, для яких оцінку $f_{th\ lim}(n)$ не доведено, а $f_{Amin}(n) \succ f_{tr}(n)$, тобто є відстань між тривіальною нижньою границею часової складності та оцінкою найбільш ефективного з існуючих алгоритмів її розв'язання. Для задач цього класу має місце теоретична проблема доведення оцінки $f_{th\ lim}(n)$ (яка може й дорівнювати $f_{tr}(n)$), що переведе задачу до класу PROP, або доведення глобальної оптимальності найбільш ефективного з існуючих алгоритмів, що переведе задачу до класу THCL. Клас THOP є доволі широким. В ньому знаходяться всі задачі з класу NPC (NP-повні задачі). До цього класу, наприклад, відноситься задача множення матриць, для якої $f_{Amin}(n) = O(n^{2,34})$, а тривіальна оцінка, що відображає необхідність перегляду всіх елементів початкових матриць $f_{tr}(n) = O(n^2)$, теоретична нижня границя часової складності для цієї задачі поки що не доведена.

Лекція 14. Класифікація алгоритмів за складністю функції працеємності. Інші спеціальні класифікації

Класифікація комп'ютерних алгоритмів на основі кутової міри асимптотичного зростання функцій [10]. В даному класифікаційному підході базуються на задачі розділу поліномів та експонент в межах однієї міри з виділенням множини функцій, що розмежовують поліноми і експоненти, та додаткових множин субполіноміальних і надекспоненціальних функцій.

Нехай: $f = f(n)$ – функція складності алгоритму, n – довжина входу алгоритму. Вважаємо, аргумент цієї функції є неперервним, просто значення функції обчислюють у цілочисельних точках $x = n$. Для розмежування поліномів і експонент використовується функція $g(x) = (\ln x)^{\ln x}$.

Твердження. Функція $g(x) = (\ln x)^{\ln x}$ розмежовує поліноми і експоненти.

Доведення. Для доведення необхідно показати, що функція $g(x)$ задовольняє за $x \to \infty$

$$\text{якщо } f(x) = x^k, k > 0, \text{ то } f(x) = o(g(x)),$$

$$\text{якщо } f(x) = e^{\lambda x}, \lambda > 0, \text{ то } g(x) = o(f(x)).$$

Для цього використаємо лему про логарифмічну границю, а саме: якщо $\lim\limits_{x\to\infty} f(x) = \infty$ та $\lim\limits_{x\to\infty} g(x) = \infty$, тоді, якщо $\lim\limits_{x\to\infty} \dfrac{\ln f(x)}{\ln g(x)} = 0$, то $\lim\limits_{x\to\infty} \dfrac{f(x)}{g(x)} = 0$, тобто $f(x) = o(g(x))$.

Отже, розглянемо $\lim\limits_{x\to\infty} \dfrac{\ln(x^k)}{\ln((\ln x)^{\ln x})} = \lim\limits_{x\to\infty} \dfrac{k\ln(x)}{(\ln x)(\ln(\ln x))} = \lim\limits_{x\to\infty} \dfrac{k}{\ln(\ln x)} = 0$,

тому $x^k = o((\ln x)^{\ln x}), k > 0$.

Далі, $\lim\limits_{x\to\infty} \dfrac{\ln((\ln x)^{\ln x})}{\ln(e^{\lambda x})} = \lim\limits_{x\to\infty} \dfrac{(\ln x)(\ln(\ln x))}{\lambda x} = 0$, тобто $(\ln x)^{\ln x} = o(e^{\lambda x}), \lambda > 0$.

Використаємо кутову міру асимптотичного зростання функцій, уведену наступним чином: задана функція $f = f(x)$, яка монотонно зростає, та $\lim\limits_{x\to\infty} f(x) = \infty$, їй у відповідність поставлена функція

$$h(x) = \ln(f(x)) + \frac{\ln(f(x))}{\ln(f(x)) + \ln x} x, \tag{14.1}$$

що має наведені нижче властивості [10]:

1. Нехай $f(x) = e^{\lambda x}(1+\gamma(x)), \lambda > 0$, $\gamma(x) = o(1), \gamma'(x) = o(1), x \to \infty$, тоді $\lim\limits_{x\to\infty} h'(x) = \lambda + 1$.

2. Нехай $f(x) = x^k(1+\gamma(x)), k > 0$, $\gamma(x) = o(1), \gamma'(x) = o(1), x\gamma'(x) = O(1)$, $x \to \infty$, тоді $\lim\limits_{x\to\infty} h(x) = \dfrac{k}{k+1}$.

3. Нехай задана функція $h(x)$ така, що $\lim\limits_{x\to\infty} h'(x) = C, C > 0$, та параметрично задана функція $z(s)$

$$z(s) = \begin{cases} s = arctg\,(1/x), \\ z = arctg\,\dfrac{1}{h(x)}, \end{cases} \qquad (14.2)$$

тоді $\lim\limits_{\substack{x\to\infty \\ (s\to 0)}} \dfrac{dz}{ds} = \dfrac{1}{C}$.

Графічно, це можна інтерпретувати наступним чином. В системі координат (z, s) поліноми та експоненти відображаються у функції, що мають в асимптотиці при $x \to \infty$, $s \to 0$ різні кути нахилу дотичної в точці $z = 0$, $s = 0$ (це й визначило назву кутової міри асимптотичного зростання функцій). Наприклад, функції z(s) отримані для $f(x) = x^2$ й $f(x) = e^x$ наведені на рис. 14.1.

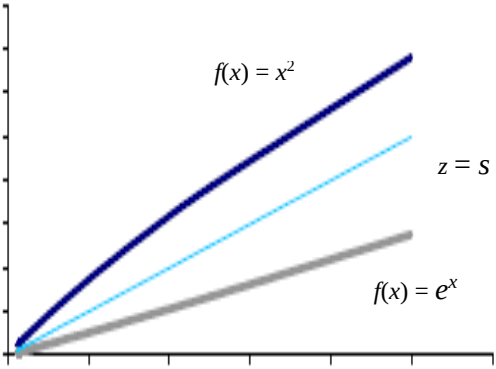


Рис. 14.1. Кутова міра асимптотичного зростання функцій x^2, e^x [10]

Теорема про кутову міру асимптотичного зростання функцій [10]. Нехай задана функція $f = f(x)$, яка монотонно зростає, та $\lim\limits_{x\to\infty} f(x) = \infty$.

Визначимо міру $\pi(f(x))$ асимптотичного (на нескінченості) зростання функції $f(x)$ як $\pi(f(x)) = \pi - 2\text{arctg}(R)$, де $R = \lim\limits_{\substack{x\to\infty \\ (s\to 0)}} \dfrac{dz}{ds}$, параметрична функція z(s) визначена у вигляді (14.2), функція $h(x)$ задана як (14.1). Тоді:

- якщо $f(x) = e^{\lambda x}(1+\gamma(x)), \lambda > 0,\ \gamma(x) = o(1), \gamma'(x) = o(1), x \to \infty,$ то $\pi/2 < \pi(f(x)) < \pi$;
- якщо $f(x) = x^k(1+\gamma(x)), k > 0,\ \gamma(x) = o(1),\ \gamma'(x) = o(1),\ x\gamma'(x) = O(1),$ $x \to \infty,$ то $0 < \pi(f(x)) < \pi/2$;
- якщо $f(x) = (\ln x)^{\ln x}$, то $\pi(f(x)) < \pi/2$.

Властивості кутової міри асимптотичного зростання функцій (дозволяють використовувати її для класифікації алгоритмів за складністю функції працеємності). Припустимо, що $\lim\limits_{x\to\infty} f(x) = \infty$.

1. Визначимо множину функцій FZ = $\{f(x) |\ f(x) \prec x^k,\ k > 0\}$. Для функції $f(x)$ з множини FZ значення R, що визначається за властивістю 3, дорівнює $+\infty$, а міра $\pi(f(x)) = \pi - 2\text{arctg}(R) = 0$ для будь-якого $f(x) \in$ FZ (наприклад, $\pi(\ln x) = 0$).
2. Визначимо множину поліномів FP $= \{f(x) |\ \exists k > 0 : f(x) = \Theta(x^k)\}$, що основується на властивості 2, але запропонована міра може бути справедливою й для більш широкого класу функцій $f(x) = \Theta(x^k)g(x)$, де $g(x) \in$ FZ. Визначимо множину функцій F_k:
$$F_k = \{f(x) |\ x^{k-\varepsilon} \prec f(x) \prec x^{k+\varepsilon},\ k > 0,\ \varepsilon \to +0, x \to \infty\},$$
тоді множину узагальнених поліномів FP можна визначити як $FP = \{f(x) | \exists k > 0 : f(x) \in F_k\}$. Для функції $f(x)$ з FP значення $R = (k+1)/k$, $k > 0$, за властивостями 2 та 3, а міра $\pi(f(x)) = \pi - 2\text{arctg}((k+1)/k)$. Тому $0 < \pi(f(x)) < \pi/2$.
3. Визначимо множину функцій FL = $\{f(x) |\ x^k \prec f(x) \prec e^{\lambda x},\ k > 0,\ \lambda > 0\}$. Для функції $f(x) \in$ FL значення R за властивістю 3 дорівнює 1, а міра $\pi(f(x)) = \pi - 2\text{arctg}(1) = \pi/2$ (наприклад, $\pi(\ln x^{\ln x}) = \pi/2$).
4. Визначимо множину експонент FE $= \{f(x) |\ \exists \lambda > 0 : f(x) = \Theta(e^{\lambda x})\}$. Уведене основується на властивості 1, але міра може бути справедливою й для більш широкого класу функцій $f(x) = \Theta(e^{\lambda x})g(x)$, де $g(x) \in$ {FZ, FP, FL}. Визначимо множину функцій F_λ
$$F_\lambda = \{f(x) |\ e^{(\lambda-\varepsilon)x} \prec f(x) \prec e^{(\lambda+\varepsilon)x}, \lambda > 0,\ \varepsilon \to +0, x \to \infty\},$$

тоді множину узагальнених експонент FE можна визначити як: FE = $\{f(x) | \ \exists\lambda > 0 : f(x) \in F_\lambda\}$. Для функції $f(x)$ з FE значення R =1/(1+ λ), λ > 0, за властивостями 1 та 3, а міра $\pi(f(x))$ = $= \pi - 2\text{arctg}(1/(1+\lambda))$, тому $\pi/2 < \pi(f(x)) < \pi$.

5. Визначимо множину функцій FF= $\{f(x) | \ e^{\lambda x} \prec f(x), \lambda > 0\}$. Для функції $f(x)$ з FF значення R за властивістю 3 дорівнює 0, а міра $\pi(f(x)) = \pi - 2\text{arctg}(R) = \pi$ (наприклад, $\pi(x^x) = \pi$).

При використанні кутової міри асимптотичного зростання функцій отримується наступна *класифікація алгоритмів за асимптотикою зростання функції працеємності* (для середнього або найгіршого випадку) [10].

1. Клас π0 (пі нуль, клас швидких алгоритмів) – до нього належать алгоритми, для яких функції працеємності належать множині FZ та мають міру 0:

$$\pi 0 = \{A | \ \pi(f_A^*(n)) = 0 \Leftrightarrow f_A^*(n) \in \text{FZ}\},$$

де $f_A^*(n)$ – функція складності для середнього або найгіршого випадку. Алгоритми, що належать до цього класу, є швидкими відносно довжини входу n, в основному це алгоритми, що мають полілогарифмічну або логарифмічну складність (наприклад, алгоритм бінарного пошуку в масиві відсортованих ключів з асимптотичною оцінкою працеємності $O(\ln n)$, його міра $\pi(\ln n) = 0$).

2. Клас πP (клас раціональних (власне поліноміальних) алгоритмів) – до нього належать алгоритми, функції складності яких з множини FP:

$$\pi\text{P} = \{A | \ 0 < \pi(f_A^*(n)) < \pi/2 \Leftrightarrow f_A^*(n) \in \text{FP}\}.$$

До цього класу відноситься більшість використовуваних на практиці алгоритмів, що дозволяють вирішувати обчислювальні задачі за раціональний час, а сам клас πP є підкласом алгоритмів, що визначають клас задач P в теорії складності обчислень.

3. Клас πL (клас субекспоненціальних алгоритмів) – до нього належать алгоритми, функції складності яких з множини FL:

$$\pi\text{L} = \{A | \ \pi(f_A^*(n)) = \pi/2 \Leftrightarrow f_A^*(n) \in \text{FL}\}.$$

До цього класу відносяться алгоритми з більше ніж поліноміальною, але менше ніж експоненціальною складністю. Алгоритми цього класу досить працеємні, а відповідні задачі звичайно належать класу NP (на-

приклад, алгоритм факторизації великих складених чисел методом узагальненого числового сита з асимптотичною оцінкою $f_A(n) = e^{O(n^{1/3}(\ln n)^{2/3})}$ та $\pi(f_A(n)) = \pi/2$ [10]).

4. Клас πE (клас власне експоненціальних алгоритмів) – до нього належать алгоритми, функції складності яких з множини FE:

$$\pi E = \{A \mid \pi/2 < \pi(f_A^*(n)) < \pi \Leftrightarrow f_A^*(n) \in FE\}.$$

До цього класу відносять алгоритми з експоненціальною працеємністю, вони реально використовуються тільки для малої довжини входу (наприклад, переборні алгоритми для точного рішення NP-повних задач: задачі про виконуваність схеми з оцінкою $O(2^n)$, задачі про кліку з оцінкою $O(n^2 2^n)$ та ін.).

5. Клас πF (клас надекспоненціальних алгоритмів) – до нього належать алгоритми, функції складності яких з множини FF:

$$\pi F = \{A \mid \pi(f_A^*(n)) = \pi \Leftrightarrow f_A^*(n) \in FF\}.$$

Це клас практично невикористовуваних алгоритмів, вони мають біль ше ніж експоненціальну працеємність (наприклад, факторіальну). До класу, наприклад, відноситься алгоритм рішення задачі комівояжера методом повного перебору з оцінкою $\Omega(n!)$ (його міра буде:

$n! = \Gamma(n+1)$, $\ln(\Gamma(x+1)) \approx x \ln x - x$, $n! \approx e^{n(\ln n - 1)}$, а $\pi(e^{x(\ln x - 1)}) = \pi$).

Перейдемо до *класифікації комп'ютерних алгоритмів за впливом на працеємність особливостей входів* [10]/ Довжина входу, конкретні значення елементів входу, їх порядок та ін., як відомо, часто впливають на ресурсні характеристики алгоритму. Тому в функції працеємності $f_A(D)$ виділимо кількісну та параметричну складову, які позначимо $f_n(n)$ та $g_p(D)$, $f_A(D) = f_A(f_n(n), g_p(D))$.

Для більшості алгоритмів функція $f_A(D)$ може бути представлена у формі $f_A(n, D) = f_n(n)\, g^*_p(D)$ або $f_A(n, D) = f_n(n) + g^+_p(D)$. Перша форма характерна для алгоритмів, коли сильно параметрично залежний зовнішній цикл, що визначає перебір варіантів рішення задачі, містить внутрішній цикл, що перевіряє рішення з кількісно залежною від розмірності працеємністю (наприклад, для алгоритмів з класу NPC). В силу скінченості множини D_n існує найгірший випадок для $g^*_p(D)$ та $g^+_p(D)$. Позначимо відповідні функції

$$g^*(n) = \max_{D \in D_n} \{g^*_p(D)\} \text{ та } g_p(n) = \max_{D \in D_n} \{g^+_p(D)\}.$$

Використаємо поняття типів алгоритмів, які визначаються довжиною входу. Це: тип L_C – алгоритми з фіксованою (сталою) довжиною входу (для них n = const, $\forall D \in D_A \left| D \right| = n$), тип L_n – алгоритми зі змінною довжиною входу (для них $\exists D_1, D_2 \in D_A \left| D_1 \right| \neq \left| D_2 \right|$).

Перейдемо до *класифікації алгоритмів за працеємністю у типі* L_C. В силу припущень типу маємо й фіксовану працеємність. Отже:

1. Клас C – містить алгоритми з фіксованою працеємністю (n = const, $\exists k = const : \forall D \in D_A \left| D \right| = n,\ f_A(D) = k$). Наприклад, алгоритм обчислення площі квадрату.

2. Клас PR – містить алгоритми з працеємністю, що параметрично залежить від значень деяких елементів множини D (n = const, $\exists k = const : \forall D \in D_A \left| D \right| = n,\ f_n(n) = const = c \Rightarrow f_A(D) = cg(D)$). Прикладом алгоритмів, що відносяться до класу PR, є алгоритми обчислення стандартних функцій із заданою точністю з обчисленням відповідних степеневих рядів.

Для класу PR, внаслідок частої наявності в алгоритмів працеємності, що залежить тільки від фіксованого числа параметрів множини початкових даних, уведений поділ на підкласи. В ньому враховується, що, за наявності лише одного параметру, що визначає працеємність, $f_A(D)$ залежить або від кількості бітів у двійковому представленні цього параметру, або від значень цих бітів (подібне отримується за умови наявності декількох таких параметрів). Тоді, позначивши m – число значущих бітів параметру, маємо $f_A(D) = f_A(m)$ та наступні підкласи.

Підклас PR pl – підклас алгоритмів з поліноміальною параметрично-залежною працеємністю ($\exists k = const : f_A(m) = O(m^k)$). До підкласу відносять алгоритми, працеємність яких залежить лише від числа бітів параметру (наприклад, алгоритм піднесення до цілого степеню методом повторного піднесення до квадрату).

Підклас PR exp – підклас алгоритмів з експоненціальною параметрично-залежною працеємністю ($\exists k = const : f_A(m) = O(e^m)$). До підкласу відносять алгоритми, працеємність яких залежить від числового значення параметрів (наприклад, алгоритм піднесення до цілого степеню методом послідовного множення).

Розглянемо *класифікацію алгоритмів за праце ємністю у типі* L_n. Для алгоритмів з типу L_n можливо декілька варіантів залежності праце ємності алгоритму від характеристичних особливостей множини *D*.

1. *Клас C* – алгоритми з фіксованою праце ємністю ($\exists k = const : \forall n : D_n \in D_A$ $\forall D \in D_n, f_A(D) = k$). Наприклад, маємо задачу пошуку максимального елементу у відсортованому масиві з *n* елементів. Тоді незалежно від значення *n* результат отримуємо за фіксоване число базових операцій моделі обчислення.

2. *Клас PR* – клас параметрично-залежних за працеємністю алгоритмів. Це алгоритми, працеємність яких визначається не вимірністю входу, а конкретними значеннями всіх або деяких елементів з вхідної множини *D* чи іншими характеристичними особливостями входу, наприклад, порядком розташування елементів ($\forall D_n \in D_A \quad \forall D \in D_n \quad f_n(n) = const = c \Rightarrow$ $f_A(n, D) = cg^*(n)$).

3. *Клас N* – клас кількісно-залежних за працеємністю алгоритмів. До нього відносять алгоритми, функція працеємності яких залежить тільки від вимірності конкретного входу ($g^+(n) = 0 \Rightarrow g^*(n) = 1 \Rightarrow f_A(D) = f_n(n)$). Прикладом є алгоритми для стандартних операцій з масивами та матрицями.

4. *Клас NPR* – клас кількісно-параметричних за працеємністю алгоритмів ($f_n(n) \neq$ const, $g^*(n) \neq$ const, $f_A(D) = f_n(n)\, g^*(n)$ або $f_A(D) = f_n(n) + g^+(n)$. Наприклад, алгоритм, що реалізує чисельний метод, в якому парамет рично-залежний зовнішній цикл по точності включає до себе кількісно-залежний фрагмент за вимірністю, породжуючи мультиплікативну форму для $f_A(D)$.

В класі PR в типі L*n* уведений поділ на підкласи.

Клас PR fix – підклас алгоритмів з працеємністю, параметрично залежною від фіксованого числа елементів множини D. Це випадок, коли лише наперед фіксоване число параметрів входу алгоритму визначає функцію працеємності.

Клас PR float – підклас алгоритмів з працеємністю, параметрично залежною від змінного числа значень елементів множини D. Наприклад, у задачі визначення числа елементів вхідного масиву, сума значень яких перевищує задане число (сумування йде у порядку зростання ін-

дексів початкового масиву). В даному прикладі працеємність визначається значеннями деякої частини елементів масиву, лише у найгіршому випадку – всіх елементів.

Розглянемо *підкласи класу NPR за степенем впливу на працеємність параметричної компоненти*, що дає можливість вивчити степінь впливу кількісної та параметричної складової на головний порядок функції $f_A(D) = f_n(n)\, g^*(n)$.

Підклас NPRL(Low) – слабо-параметричний підклас класу NPR ($g^+(n) = O(f_n(n)) \Leftrightarrow g^*(n) = \Theta(1)$). Для таких алгоритмів параметрична складова впливає не більше ніж на коефіцієнт головного порядку функції працеємності. Прикладом є алгоритм пошуку максимуму у масиві (кількість переприсвоєнь максимуму у найгіршому випадку, що визначає $g^+(n) = \Theta(n)$, має такий же порядок як і оцінка зовнішнього циклу перебору елементів).

Підклас NPRE(Equivalent) – середньо-параметричний підклас класу NPR, алгоритми, у яких складова $g^*(n)$ функції працеємності має порядок зростання, що не перевищує $f_n(n)$ ($g^*(n) = \Theta(f_n(n) \Leftrightarrow g^+(n) = \Theta(f_n^2(n))$). У таких алгоритмів параметрична складова має вплив близький за силою впливу кількісної компоненти на головний порядок функції праце ємності. Прикладом є алгоритм сортування масиву методом бульбашки, для якого кількість перестановок елементів у гіршому випадку (масив відсортований в оберненому порядку) визначає $g^+(n) = \Theta(n^2)$, а $f_n(n) = \Theta(n)$.

Підклас NPRH(High) – сильно-параметричний підклас класу NPR, алго ритми, у яких складова $g^*(n)$ функції працеємності має асимптотичний порядок зростання вище $f_n(n)$ ($f_n(n) = o(g^*(n))$). В таких алгоритмах саме параметрична складова визначає головний порядок функції працеємності. Прикладом є алгоритми точного рішення NP-повних задач.

Розглянемо *підкласи класу NPR за характеристичними особливостями множини початкових даних.* Такий поділ на підкласи передбачає виділення у функції $g_p(D)$ компонент виду $g_p(D) = g_v(D) + g_s(D)$, де $g_v(D)$ – компонента, повязана із значеннями елементів входу, $g_s(D)$ – компонента, пов'язана з порядком елементів входу. Виділимо у $D \in D_n$ підмножину однорідних за суттю задачі елементів D_e, що складається з m елементів $\{d_1, d_2, \ldots, d_m\}$. Визначимо D_p як множину всіх впорядкованих послідовностей з $\{d_1, d_2, \ldots, d_m\}$ ($|D_p| = m!$). Якщо $D_{e1}, D_{e2} \in D_p$,

то відповідні їм D_1, $D_2 \in D_n$. Залежність працеємності від значень елементів входу має на увазі, що $f_A(n, D) = f_A(n, p_1, \ldots, p_m), m \le n$, $p_i \in D$, $i = \overline{1, m}$.

Підклас NPRS(Sequense) – підклас алгоритмів з кількісною та порядково-залежною функцією працеємності: $g_v(D) = o(g_s(D))$. В цьому випадку працеємність залежить від вимірності входу та від порядку розміщення однорідних елементів, залежність від значень не є суттєвою. Прикладом є більшість алгоритмів сортування порівняннями, алгоритми пошуку мінімуму та максимуму в масиві.

Підклас NPRV(Value) – підклас алгоритмів з функцією працємності, залежною від довжини входу і значень елементів у D ($g_s(D) = o(g_v(D))$). Прикладом є алгоритм рішення задачі пакування методом динамічного програмування (табличний метод), в якому функція працеємності залежить від кількості типів вантажів та від значень обсягів вантажів та пакування. Порядок оброблення вантажів не є визначаючим.

Але *NPR* – (*NPRS* $\cup$ *NPRV*) $\neq \emptyset$ так як існують алгоритми, в яких значення та порядок розміщення однорідних елементів здійснюють суттєвий вплив на функцію працеємності (наприклад, ітераційні алгоритми рішення систем лінійних рівнянь – перестановка значень у початковій матриці та зміна цих значень суттєво змінюють власні числа матриці, які визначають збіжність ітераційного процесу розв'язання).

Класифікація алгоритмів за інформаційною та вимірнісною чуттєвістю [10]. Ідеалом прогнозування працеємності алгоритму є отримання точної функції $f_A(D)$. Але це можливо тільки для кількісно-залежних алгоритмів (клас *N*) та для окремих алгоритмів інших класів. Для більшості алгоритмів класу *NPR* одержати точну функцію працеємності важко навіть для відносно простих алгоритмів, використання ж оцінки у середньому дозволяє прогнозувати працеємність лише при усередненні за великою вибіркою входів.

Більш детальний розгляд задачі прогнозування пов'язаний з вивченням впливу характеристик різних початкових даних на функцію працеємності. Звичайно вплив змін параметрів входу на вихідну характеристику досліджуваного об'єкту називають *чуттєвістю за вхідним параметром*. Тому розглянемо чуттєвість функції працеємності алгоритму до початкових даних. А т.я. працеємність алгоритму залежить від кількості елементів множини початкових даних та від параметрів цієї мно-

жини, то можна виділити різні аспекти чуттєвості алгоритмів. Під вимірнісною чуттєвістю алгоритму розуміють вплив зміни вимірності на значення функції працеємності. Під інформаційною чуттєвістю алгоритму розуміють вплив різних входів фіксованої вимірності на зміну значень функції працеємності алгоритму.

Нехай маємо деякий алгоритм з класу *NPR* та його функцію працеємності $f_A(n)$, а також можемо представити графік залежності $f_A(D \in D_n)$ від конкретних входів алгоритму з вимірністю n, всього таких входів $|D_n|$. Так як маємо справу з цілочисельною функцією цілочисельного аргументу, графік буде множиною точок, причому для деякого входу алгоритм буде виконувати максимальну кількість операцій (маємо найгірший випадок, $f_A^{\wedge}(n)$), а для деякого – мінімальну (маємо найкращий випадок, $f_A^{\vee}(n)$). Всі інші точки на графіку будуть знаходитися між мінімальним та максимальним значеннями. Підраховуючи відносні за вимірністю множини D_n частотні зустрічі значень функції працеємності f_A можна отримати відносні частоти значень функції працеємності $P(f_A)$ як дискретної випадкової величини, причому $\sum P(f_A) = 1$. Тоді середнє значення працеємності: $\bar{f}_A(n) = \sum f_A P(f_A)$. Розглядаючи функцію працеємності як дискретну випадкову величину, що характеризується математичним очікуванням й дисперсією та обмежену максимальним й мінімальним значеннями, можна отримати більш детальну інформацію про поведінку алгоритму. Іноді математичне очікування та дисперсія можуть бути одержані теоретично, на основі аналізу входів фіксованої довжини. Позначимо інформаційну чуттєвість $\delta_i(n)$. Тоді має місце наступне.

Теорема [10]. Неперервна випадкова величина, обмежена по x сегментом $[a,b]$, й задана на ньому функцією розподілу $f(x)$ ($\forall x : x \in [a,b]: f(x) \geq 0,$ та $\int_a^b f(x)dx = 1$) має максимальну дисперсію $D[f(x)]$, якщо функція $f(x)$ є вигляду $f(x) = q\delta(x-a) + p\delta(x-b)$, причому $p = q = ½$, де $\delta(x)$ – дельта функція, при цьому $D[f(x)] \leq 1/4(a-b)^2$.

Наведена теорема дозволяє стверджувати, що дисперсія σ_{f_A} буде максимальною тоді, коли функція працеємності як дискретна випадкова

величина приймає з рівною ймовірністю тільки мінімальне або максимальне значення. Тоді математичне очікування та дисперсія:

$$E[f_A(n)] = (f_A^{\wedge}(n) + f_A^{\vee}(n))/2, \quad \sigma_{f_A \max}(n) = (f_A^{\wedge}(n) - f_A^{\vee}(n))/2.$$

Врахуємо ще розмір інтервалу можливих значень функції працеємності. За однакового значення дисперсії більш чуттєвим є алгоритм з більшим інтервалом можливих значень. Використаємо поняття розмаху варіювання ($R = (f_A^{\wedge}(n) - f_A^{\vee}(n))$) та поняття нормованого (відносного) розмаху варіювання функції працеємності для входів довжини n ($R_N(n) = (f_A^{\wedge}(n) - f_A^{\vee}(n))/(f_A^{\wedge}(n) + f_A^{\vee}(n))$) [10]. Оскільки всі значення функції працеємності додатні та працеємність у найгіршому випадку не менша, ніж у найкращому ($f_A^{\wedge}(n) \geq f_A^{\vee}(n)$), то $0 \leq R_N(n) \leq 1$.

Використання генерального коефіцієнту варіації V для функції працеємності як функції довжини входу має вигляд $V(n) = \sigma_{f_A(n)} / \overline{f}_A(n)$, $0 \leq V(n) \leq 1$, де $\sigma_{f_A(n)}$ – середньоквадратичне відхилення функції працеємності (як дискретної величини) за фіксованої вимірності входу n. У найгіршому випадку для дисперсії функції працеємності генеральний коефіцієнт варіації співпадає з нормованим розмаху варіювання, $\sigma_{f_A(n)}$ може змінюватися від 0 до $\sigma_{f_A \max}(n)$. Уведення кількісної міри інформаційної чуттєвості алгоритму як $\delta_i(n) = V(n) R_N(n)$, $0 \leq \delta_i(n) \leq 1$, дозволяє в інформаційній чуттєвості алгоритму за фіксованої довжини входу враховувати середнеквадратичне відхилення функції працеємності та розмір інтервалу можливих значень у нормованих одиницях.

Наведене може бути використаним для детального дослідження алгоритмів. Наприклад, кількісна міра інформаційної чуттєвості може бути застосована для більш обґрунтованого рішення задачі вибору раціональних алгоритмів за ресурсною ефективністю. Якщо до програмної системи висунуті часові вимоги з вузькими границями, раціональним є вибір алгоритму з малою кількісною мірою інформаційної чуттєвості. Крім того це є основою для класифікації алгоритмів за вказаною кількісною мірою інформаційної чуттєвості.

Розглянемо вимірнісну чутливість для найкращого, середнього та найгіршого випадків функції працеємності за фіксованої довжини входу

($f_A^*(n)$). Тоді маємо відповідно три кількісних міри вимірнісної чутливості, що позначимо $\delta_n^*(n)$.

Нехай функція працеємності $f_A^*(n)$ задана рекурсивно: $f_A^*(n+1) = h_A^*(n) f_A^*(n)$. Т.я. функція $f_A^*(n)$ є неспадною, то $f_A^*(n+1) \geq f_A^*(n)$ й $\lim\limits_{n\to\infty} f_A^*(n) = \infty$, то $h_A^*(n) \geq 1$.

Представимо $h_A^*(n)$ як $h_A^*(n) = 1 + \delta_n^*(n)$, $\delta_n^*(n) \geq 0$. Тоді $\delta_n^*(n) = \dfrac{f_A^*(n+1) - f_A^*(n)}{f_A^*(n)}$, а рекурсивно задана функція $f_A^*(n)$ зв'язана з мірою вимірнісної чутливості наступним чином:

$$f_A^*(n+1) = f_A^*(n) + \delta_m^*(n) f_A^*(n). \qquad (14.3)$$

Розглянемо поведінку міри вимірнісної чутливості для різних підкласів алгоритмів у класі *NPR* (як найбільш широкому класі практично використовуваних алгоритмів). Нехай $f_n(n)$ – кількісна компонента функції працеємності, $g^+(n)$ – параметрична компонента в адитивній формі, $g^*(n)$ – параметрична компонента в мульпликативній формі.

Підклас NPRL ($g^+(n) = O(f_n(n))$). Вважаючи, що $g^+(n) \leq c\, f_n(n)$, для алгоритмів підкласу кількісна міра вимірнісної чутливості:

$$\delta_n^*(n) = \frac{f_n(n+1) + cf_n(n+1) - f_n(n) - cf_n(n)}{f_n(n) + cf_n(n)} = \frac{f_n(n+1) - f_n(n)}{f_n(n)}.$$

Тому для алгоритмів підкласу кількісна міра вимірнісної чутливості працеємності співпадає з мірою її кількісної компоненти.

Підклас NPRE ($g^*(n) = \Theta(f_n(n))$). Вважаючи, що $g^*(n) = cf_n(n)$, для алго ритмів підкласу кількісна міра вимірнісної чутливості:

$$\delta_n^*(n) = \frac{f_n(n+1)cf_n(n+1) - f_n(n)cf_n(n)}{f_n(n)cf_n(n)} = \frac{f_n(n+1) - f_n(n)}{f_n(n)} \frac{f_n(n+1) + f_n(n)}{f_n(n)} =$$

$$= \delta_{f_n}(n)\left(1 + \frac{f_n(n+1)}{f_n(n)}\right),$$

де $\delta_{f_n}(n)$ – міра вимірнісної чуттєвості кількісної компоненти функції працеємності. Т.я. $f_n(n+1) \geq f_n(n)$, то $\delta_n^*(n) \geq 2\delta_{f_n}(n)$, тобто для алгоритмів підкласу кількісна міра вимірнісної чуттєвості не менше ніж у 2 рази перевищує кількісну міру вимірнісної чуттєвості кількісної компоненти функції працеємності.

Підклас NPRH ($f_n(n) = o(g^*(n))$). Використовуючи рекурсивну залежність (14.3) маємо: $f_n(n+1) = f_n(n) + \delta_{f_n}(n) f_n(n)$, $g^*(n+1) = g^*(n) + \delta_g^*(n) g^*(n)$, де $\delta_g^*(n)$ – міра вимірнісної чуттєвості параметричної компоненти функції працеємності. Тоді для алгоритмів підкласу кількісна міра вимірнісної чутливості:

$$\delta_n^*(n) = \frac{(f_n(n) + \delta_{f_n}(n) f_n(n))(g^*(n) + \delta_g^*(n) g^*(n)) - f_n(n) g^*(n)}{f_n(n) c f_n(n)} =$$

$$= \delta_{f_n}(n) + \delta_g^*(n) + \delta_{f_n}(n) \delta_g^*(n) = \delta_g^*(n)(1 + \delta_{f_n}(n) + \frac{\delta_{f_n}(n)}{\delta_g^*(n)}).$$

А так як для підкласу $\delta_g^*(n) >> \delta_{f_n}(n)$, то $\delta_n^*(n) \approx \delta_g^*(n)(1 + \delta_{f_n}(n))$, тобто для алгоритмів підкласу кількісна міра вимірнісної чуттєвості визначається параметричною та кількісною компонентами функції працеємності.

Для уведення *класифікації алгоритмів за інформаційною чутливістю* розглянемо основні випадки, коли функція працеємності як дискретна випадкова величина відповідає різним законам розподілу ймовірностей [10].

1. Функція працеємності має максимальне середньоквадратичне відхилення, а розмах варіювання прямує до 1:

$$\sigma_{f_A}(n) = \sigma_{f_A \max}(n) = (f_A^\wedge(n) - f_A^\vee(n))/2, \quad \overline{f}_A(n) = (f_A^\wedge(n) + f_A^\vee(n))/2,$$

$f_A^\vee(n) << f_A^\wedge(n)$ або $f_A^\vee(n) = o(f_A^\wedge(n))$, що забезпечує $R_N(n) \to 1$ при $n \to \infty$, а тоді $\delta_i(n) \to 1$ при $n \to \infty$.

2. Гістограма відносних частот функції працеємності відповідає рівномірному закону розподілення, $R_N(n) \to 1$ при $n \to \infty$:

$$\sigma_{f_A}(n) = (f_A^\wedge(n) - f_A^\vee(n))/\sqrt{12}, \quad \overline{f}_A(n) = (f_A^\wedge(n) + f_A^\vee(n))/2,$$

тому $\delta_i(n) \to 2\sqrt{12}$ при $n \to \infty$.

3. Гістограма відносних частот функції працеємності відповідає нормальному закону розподілення, $R_N(n) \to 1$ при $n \to \infty$, а розмах варіювання складає $6\sigma_{f_A}(n)$:

$$\sigma_{f_A}(n) = (f_A^\wedge(n) - f_A^\vee(n))/6, \quad \overline{f}_A(n) = (f_A^\wedge(n) + f_A^\vee(n))/2,$$

тому $\delta_i(n) \to 2/6 = 1/3$ при $n \to \infty$.

4. Гістограма відносних частот функції працеємності відповідає нормальному закону розподілення, $R_N(n) \to 1$ при $n \to \infty$, а середньоквадратичне відхилення складає 1/40 (2.5%) від розмаху варіювання (тобто інтервал $2\sigma_{f_A}(n)$ складає 5% від цього розмаху):

$$\sigma_{f_A}(n) = (f_A^{\wedge}(n) - f_A^{\vee}(n))/40, \ \overline{f}_A(n) = (f_A^{\wedge}(n) + f_A^{\vee}(n))/2,$$

тому $\delta_i(n) \to 2/40 = 0.05$ при $n \to \infty$.

5. Функція працеємності має нульове середньоквадратичне відхилення (працеємність для будь-яких входів фіксованої довжини однакова), дисперсія та розмах варіювання дорівнюють 0, тому $\delta_i(n) = 0$.

За наведеними випадками існує така класифікація [10].

Клас i1. Алгоритми, що не чутливі до вхідних даних по функції працеємності за фіксованої довжини входу, $\delta_i(n) = 0$. Цей клас співпадає з класом кількісно-залежних алгоритмів (клас *N*).

Клас i2. Алгоритми, що є слабо чутливими до вхідних даних по функції працеємності за фіксованої довжини входу, $0 < \delta_i(n) < 0.05$.

Клас i3. Алгоритми, що є чутливими до вхідних даних по функції працеємності за фіксованої довжини входу, $0.05 < \delta_i(n) < 1/3$.

Клас i4. Алгоритми, що є сильно чутливими до вхідних даних по функції працеємності за фіксованої довжини входу, $1/3 < \delta_i(n) \le 1$.

Слід відзначити, що оскільки кількісна міра інформаційної чутливості $\delta_i(n)$ є комплексною мірою, то однакову чутливість можуть мати алгоритми, що мають малих розмах варіювання за більшої дисперсії та алгоритми з більшим розмахом варіювання за меншої дисперсії. Такий підхід є раціональним, т.я. в кожному з цих випадків розкид (за ймовірністю) очікуваних відносних змін функції працеємності буде приблизно однаковим. Крім того, оскільки $\delta_i(n)$ залежить від вимірності, то може статися, що один й той самий алгоритм за різних вимірностей буде відноситися до різних типів за інформаційною чутливістю.

Існують два підходи до визначення міри $\delta_i(n)$: теоретичний та дослідний. За теоретичного підходу потрібно одержати функції працеємності

для найкращого, середнього та найгіршого випадків, теоретичне середньоквадратичне відхилення $\sigma_{f_A}(n)$, що дає знайти $\delta_i(n)$ у вигляді функції. Далі, аналізуючи функцію $\delta_i(n)$ та обчислюючи її значення для інтервалу вимірностей, що характеризують область застосування, визначають значення інформаційної чутливості та належність алгоритму до вказаних вище класів. За дослідного підходу отримують статистичне розподілення вибірки, на основі якого обчислюють необхідні для визначення $\delta_i(n)$ величини. В цьому випадку на основі серії дослідів за фіксованої вимірності та використовуючи вибіркове середнє, вибіркову дисперсію і вибірковий коефіцієнт варіації прогнозують очікувану працеємність на основі значень $\delta_i(n)$. За такого підходу отримують вибіркову інформаційну чутливість.

Розглянемо *класифікацію алгоритмів за вимірнісною чутливістю* та приклади визначення вимірнісної чутливості.

Нехай маємо функцію працеємності $f_A^*(n)$ поліноміального вигляду. Її асимптотична головна компонента є степеневою функцією, $f_A^*(n) = cn^k$, $k > 0$ ціле, $c > 0$. Тоді за визначенням $\delta_n^*(n)$:

$$\delta_n^*(n) = \frac{c(n+1)^k - cn^k}{cn^k} = \frac{ckn^{k-1} + O(n^{k-2})}{cn^k} = \frac{k}{n} + O(n^{-2}).$$

Аналогічне отримується й за дійсного $k > 0$, якщо замінити наближено $f_A^*(n+1)$ диференціалом $f_A^*(n)$ за $\Delta n = 1 - f_A^*(n+1) \approx f_A^*(n) + kn^{k-1}\Delta n$. Тому головний порядок вимірнісної чуттєвості для степеневих функцій працеємності має вигляд k/n, не залежить від сталої c та гіперболічно зменшується із зростанням вимірності (наприклад, для $f_A^*(n) = cn$ $\delta_n^*(n) = 1/n$).

Якщо функція працеємності експоненціального вигляду, $f_A^*(n) = ce^{\lambda n}$, $\lambda > 0$, $c > 0$. Тоді за визначенням $\delta_n^*(n)$:

$$\delta_n^*(n) = \frac{ce^{\lambda(n+1)} - ce^{\lambda n}}{ce^{\lambda n}} = \frac{ce^{\lambda n}e^{\lambda} - e^{\lambda n}}{cn^k} = e^{\lambda} - 1 = const > 0.$$

Клас n1 – клас алгоритмів, що дуже мало чуттєві (не більш за лінійну функцію) до змін вимірності множини початкових даних по функції працеємності ($0 < \delta_n^*(n) < n^{-1}, \forall n > 1$).

Клас n2 – клас алгоритмів, що мало чуттєві (не більш за степеневу функцію із степенем більше 1) до змін вимірності множини початкових даних по функції працеємності ($\delta_n^*(n) = kn^{-1}, \forall n > 1, k > 1$).

Клас n3 – клас алгоритмів, що чуттєві (більш за степеневу функцію, але менше за показникову) до змін вимірності множини початкових даних по функції працеємності ($\delta_n^*(n) \succ n^{-1}$, $\lim\limits_{n\to\infty} \delta_n^*(n) = 0$).

Клас n4 – клас алгоритмів, що сильно чуттєві (не менше за показникову функцію) до змін вимірності множини початкових даних по функції працеємності ($\delta_n^*(n) = const$, $\lim\limits_{n\to\infty} \delta_n^*(n) = \infty$).

Відзначимо, що один й той самий алгоритм з функцією працеємності для найкращого, середнього та найгіршого випадків може мати різну кількісну міру та різні типи вимірнісної чутливості.

Розглянемо *класифікацію за вимогами до додаткової пам'яті*, а саме за потрібного алгоритму обсягу додаткової оперативної пам'яті у прийнятій моделі обчислень (інші види пам'яті не розглядаємо) [10].

|Нехай D – конкретний вхід алгоритму A, $|D| = n$. Ресурсні вимоги алгоритму до пам'яті визначаються пам'яттю входу, виходу і додатковою, що задіяні алгоритмом у ході його виконання.

Позначимо $V_I(D)$ таку компоненту $V_A(D)$ як пам'ять входу. Розрізнятимемо алгоритми із статичним (наперед визначеним) входом, що зберігають вхідні дані у пам'яті повністю ($V_I(D) = V_I(n)$), та алгоритми з потоковим входом (об'єкти входу поступають та обробляються алгоритмом поелементно, але поелементне зчитування об'єктів входу з файлу у визначений масив є непотоковим входом), коли для зберігання поточного об'єкту потрібно не більше за фіксовану кількість комірок оперативної пам'яті ($V_I(D) = \Theta(1)$).

Таку компоненту $V_A(D)$ як пам'ять виходу позначимо $V_R(D)$. Розрізнятимемо алгоритми із статичним (наперед визначеним) виходом, що зберігають результат у пам'яті повністю (може використовуватися спеціальний блок пам'яті або пам'ять входу (тоді $V_R(D) = 0$, наприклад, алгоритм сортування на місці), та алгоритми з потоковим виходом (об'єкти виходу поступають поелементно на деякий пристрій виводу, тоді потрібно пам'яті не більш ніж для одного елементу результату, $V_R(D) = \Theta(1)$).

Позначимо $V_t(D)$ таку компоненту $V_A(D)$ як додаткову пам'ять. Саме ця компонента розрізняє алгоритми за ресурсними витратами пам'яті в межах статичного або потокового типів.

Клас V0 – алгоритми з нульовою додатковою пам'яттю ($\forall n > 0, \forall D \in D_n$ $V_t(D) = 0$). Алгоритми цього класу або не вимагають додаткових комірок пам'яті, або використовують ресурси пам'яті входу або/та виходу.

Клас VC – алгоритми з фіксованою додатковою пам'яттю ($\forall n > 0$, $\forall D \in D_n$ $V_t(D) = const \neq 0$). Алгоритми цього класу використовують сталe число додаткових комірок оперативної пам'яті (не залежить від довжини та особливостей входу й довжини виходу). Наприклад, комірки для зберігання лічильників циклів, проміжних результатів обчислень.

Клас VL – алгоритми, додаткова пам'ять яких лінійно залежить від довжини входу. Використаємо функцію $V_t^{\wedge}(n)$ додаткової пам'яті у найгіршому випадку для всіх припустимих входів з мірою n, $V_t^{\wedge}(n) = \max\limits_{D \in D_n}\{V_t(D)\}$.

Тоді для алгоритмів класу $V_t^{\wedge}(n) = \Theta(n)$.

Клас VQ – алгоритми, додаткова пам'ять яких квадратично залежить від довжини входу. Для даного типу алгоритмів величина додаткової пам'яті у найгіршому випадку для всіх припустимих входів з мірою n пропорційна за порядком квадрату міри вимірності: $V_t^{\wedge}(n) = \Theta(n^2)$. Прикладом є стандартний алгоритм множення довгих чисел, що задані побітно масивами довжиною n.

Клас VP – алгоритми, додаткова пам'ять яких поліноміально надквадратично залежить від довжини входу. Для даного класу алгоритмів: $\exists k > 2 : V_t^{\wedge}(n) = \Theta(n^k)$. Прикладом є рекурсивні алгоритми, що породжують дерево рекурсії з оцінкою глибини $V_t^{\wedge}(n) = \Theta(n^k), k > 1$ та вимагають збереження у кожній вершині дерева додаткового масиву з довжиною порядку $\Theta(n)$.

Клас VE – алгоритми, додаткова пам'ять яких екпоненціально залежить від довжини входу. Для даного класу алгоритмів: $\exists \lambda > 2 : V_t^{\wedge}(n) = \Theta(e^{\lambda n})$. Прикладом є алгоритми, що використовуютьдодаткові масиви або більш складні структури даних, розмір яких визначається значеннями елементів входу (алгоритм сортування методом індексів або табличні алгоритми для реалізації методу динамічного програмування).

Лекція 15. Порівняльний аналіз та раціональний вибір комп'ютерних алгоритмів

Раціональний вибір комп'ютерного алгоритму для розв'язання задачі вимагає проведення дослідження алгоритмів-претендентів не тільки для отримання асимптотичних оцінок складності алгоритмів. Часто для обчислювальних задач асимптотично більш ефективний алгоритм має доволі великий коефіцієнт при головному порядку функції працеємності й тому не завжди є ефективним на всьому досліджуваному діапазоні вимірності входу (реальні значення діапазону відомі з аналізу області застосування, тобто визначаються особливостями використання алгоритму у розроблюваній програмній системі). Подібне може поширюватися й на функцію ресурсної ефективності алгоритму. Для вибору раціонального алгоритму в термінах ресурсної ефективності маємо детально дослідити алгоритми-претенденти в межах реального діапазону застосування. Т.я. не завжди алгоритм, що має асимптотично оптимальну складність (у смислі оцінок O або Θ), має найкращі значення функції працеємності у цілочисельних точках значень вимірності реальної множини початкових даних внаслідок суттєвої різниці констант, що сховані за O або Θ асимптотичними оцінками різних алгоритмів-претендентів, то в результаті на різних інтервалах цього діапазону можуть бути вибраними різні алгоритми як найбільш раціональні й доцільним буде адаптивне за вимірністю входу керування вибором алгоритму рішення задачі у програмній системі, що розроб ляється. Часто асимптотично краща продуктивність досягається за рахунок потрібного більшого об'єму додаткової пам'яті та погіршення коефіцієнту не тільки у першої, а й у другої адитивної компоненти функції працеємності. Наприклад, порівняємо функції працеємності для середнього випадку алгоритму сортування вставками ($A1$) та алгоритму сортування злиттям ($A2$):

$$\overline{f}_{A1}(n) = 2.5n^2 + 11.5n - 10,\ \overline{f}_{A2}(n) = 18n \lg n + 85n - 66.$$

Кутова міра близькості ресурсних оцінок. Для порівняльного аналізу ресурсних функцій комп'ютерних алгоритмів уведена спеціальна міра розходження значень цих функцій за заданої вимірності n (позначають $\pi(f(n), g(n))$) [10]. Міра розходження значень функцій за фіксованого значення n має задовольняти наступним вимогам:

- антисиметричності, $\pi(f(n), g(n)) = -\pi(g(n), f(n))$ (звідси випливає: $\pi(f(n), g(n)) = 0$, якщо $f(n) = g(n)$);
- можливості кутової інтерпретації значення міри $\pi(f(n), g(n))$ (випливає обмеженість значень міри за будь-яких значень аргументів, $|\pi(f(n), g(n))| \le const, \forall n > 0$).

Використаємо в якості міри розходження значень двох функцій (f та g) у точці $n = n_1$ функцію вигляду

$$\pi(a,b) = arctg\frac{a}{b} - arctg\frac{b}{a}, \text{ де } a = f(n_1), b = g(n_1).$$

Значення такої кутової міри інтерпретують як «кутове розходження» значень аргументів, що відповідають довжинам катетів прямокутного трикутника відносно значення $\pi/4$. Оскільки значення ресурсних функцій строго додатні для будь-якого n, матимемо: $-\pi/2 < \pi(f(n), g(n)) < \pi/2$. Також міра має властивість антисиметричності та стає рівною 0 при рівності значень аргументів. **Порогове значення φ**, за якого розходження значень функцій є значущим, може бути інтерпретоване як максимальне значення «кута розходження» значень двох ресурсних функцій, за якого алгоритми можуть бути визнані як однаково застосовувані на досліджуваному інтервалі вимірностей входу.

Порівняльний аналіз алгоритмів за ресурсними функціями.

Для прийняття рішення з вибору раціонального алгоритму необхідно:
- проаналізувати ресурсну ефективність порівнюваних алгоритмів (отримати функції працеємності, функції об'єму пам'яті); виконати порівняльний аналіз ресурсних функцій алгоритмів-претендентів з метою вибору раціонального алгоритму рішення задачі при реальних обмеженнях на вимірність множини початкових даних, значення тих характеристик реальної множини початкових даних задачі (D_A), за яких перевага може бути надана одному з аналізованих алгоритмів (вважатимемо, що це вимірність множини початкових даних, n).

Для класу кількісно-залежних алгоритмів (клас N) працеємність алгоритму (точного значення) повністю визначається вимірністю множини початкових даних $f_A(D \in D_n) = f_A(n)$. Для підкласів кількісно-параметричних алгоритмів із слабим та середнім параметричним впливом (підкласи *NPRL, NPRE*) в якості аналізованої функції працеємності може бути середнє значення $\bar{f}_A(n)$ за умови малої інформаційної чутливості. Для алгоритмів з кількісно-параметричного підкласу *NPRH* в

якості аналізованої функції працеємності може бути середнє значення $\bar{f}_A(n)$ або функція працеємності у найгіршому випадку.

Порівняльний аналіз ресурсних функцій передбачає визначення практично значущих інтервалів *n*, за яких даний алгоритм може бути обраним як раціональний.

Початковими даними для аналізу ресурсних функцій на скінченому інтервалі є *відомі ресурсні функції* алгоритмів (працеємності, об'єму пам'яті), *результатами – переважні інтервали характеристичних значень множини початкових даних* задачі для раціонального застосування деякого алгоритму. На практиці вибір деякого алгоритму, як переважного, може бути зробленим навіть тоді, коли його ресурсна функція, наприклад, працеємність на даному інтервалі, дещо гірша, але *не більш ніж на поріг φ* у порівнянні з працеємністю інших алгоритмів. Може статися, що на інтервалі й декілька алгоритмів використовуються еквівалентно з розходженням ресурсних функцій не більше ніж на поріг φ.

Нехай функції працеємності двох алгоритмів задані в явному вигляді функціями $f(n)$ і $g(n)$ (як пара точних функцій працеємності або функцій працеємності алгоритму для середнього чи найгіршого випадку), заданий інтервал (a, b) значень *n*, поріг φ припустимого розходження значень функцій на заданому інтервалі й деяка міра $\pi(f(n), g(n))$ розходжения значень функцій за заданого значення *n*. Тоді:

$$f(n) = \Delta_\varphi(g(n)) \text{ якщо } \min\{\pi(f(n), g(n))\} \geq \varphi \text{ для } \forall n = (a,b);$$

$$f(n) = \Theta_\varphi(g(n)) \text{ якщо } \max|\{\pi(f(n), g(n))\}| < \varphi \text{ для } \forall n = (a,b);$$

$$f(n) = o_\varphi(g(n)) \text{ якщо } \max|\{\pi(f(n), g(n))\}| \leq -\varphi \text{ для } \forall n = (a,b).$$

Тому функція $f(n) \in \Delta_\varphi(g(n))$ на інтервалі (a, b) з порогом φ, якщо значення міри $\pi(f(n), g(n))$ в усіх точках цього інтервалу більше або дорівнює φ. Функція $f(n) \in \Theta_\varphi(g(n))$ на інтервалі (a, b) з порогом φ, якщо абсолютні значення міри $\pi(f(n), g(n))$ в усіх точках цього інтервалу менше φ (тобто функції різняться менше ніж на поріг φ). Функція $f(n) \in o_\varphi(g(n))$ на інтервалі (a, b) з порогом φ, якщо значення міри $\pi(f(n), g(n))$ в усіх точках цього інтервалу менше або дорівнює пороговому значенню φ з від'ємним знаком (є протилежним $\Delta_\varphi(g(n))$ – з $f(n) = o_\varphi(g(n))$ випливає, що $g(n) = \Delta_\varphi(f(n))$).

При виконанні аналізу ресурсних функцій алгоритмів на скінченому інтервалі:

1) отримують в явному вигляду ресурсні функції алгоритмів для середнього або найгіршого випадку, або точні функції для алгоритмів класу N;
2) визначають інтервал або сегмент зміни вимірності множини початкових даних (n_1, n_2) на основі особливостей застосування даного алгоритму в умовах конкретної програмної системи;
3) розбивають (n_1, n_2) на підінтервали, для яких в кожній цілочисельній точці явно виконується одне з наведених співвідношень для прийнятої в інтервальному аналізі міри $\pi(f(n), g(n))$:

- якщо $\varphi - \pi(f(n), g(n)) \le 0$, то $f(n) = \Delta_{\varphi}(g(n))$ й переважним на даному підінтервалі є алгоритм з функцією працеємності $g(n)$;
- якщо $|\pi(f(n), g(n))| - \varphi < 0$, то $f(n) = \Theta_{\varphi}(g(n))$ й обидва алгоритми з точністю до порогу φ можуть бути однаково застосованими на підінтервалі;
- $\pi(f(n), g(n)) + \varphi \le 0$, то $f(n) = \Delta_{\varphi}(g(n))$ й переважним на підінтервалі є алгоритм з функцією працеємності $f(n)$.

На основі розбиття інтервалу (n_1, n_2) на підінтервали й приймається рішення про вибір раціонального за ресурсною функцією алгоритму.

Графічно інтерпретувати наведені результати можна так (рис. 15.1 [10]):

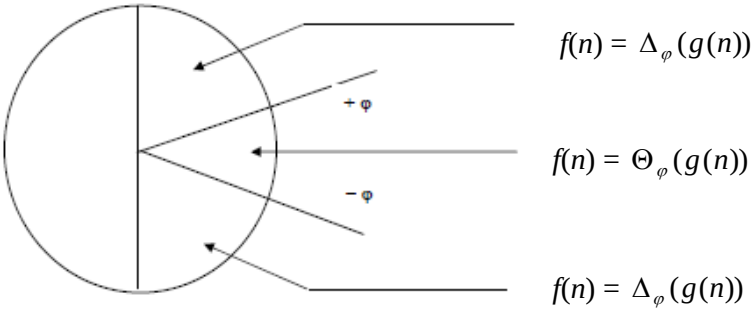


Рис. 15.1. Графічна ілюстрація вибору раціонального алгоритму з використанням кутової міри $\pi(f(n), g(n))$

Наприклад, маємо функції працеємності для середнього випадку для двох алгоритмів сортування: для алгоритму сортування методом пошуку максимуму та мінімуму з працеємністю у середньому – $f(n) = 1{,}75n^2 + 2\,n\ln n + 15n + 9$; для алгоритму сортування методом злиття з працеємністю у середньому – $g(n) = 18n\,\lg n + 85n - 66$. Для алгоритмів виберемо інтервал $(n_1, n_2) = (71, 99)$ (графіки функцій працеємності на рис. 15.2), щоб за порогового значення $\varphi = \pi/32 \approx 0{,}0981$ в ньому містилася б область, в якій виконується $f(n) \in \Theta_{\varphi}(g(n))$ [10].

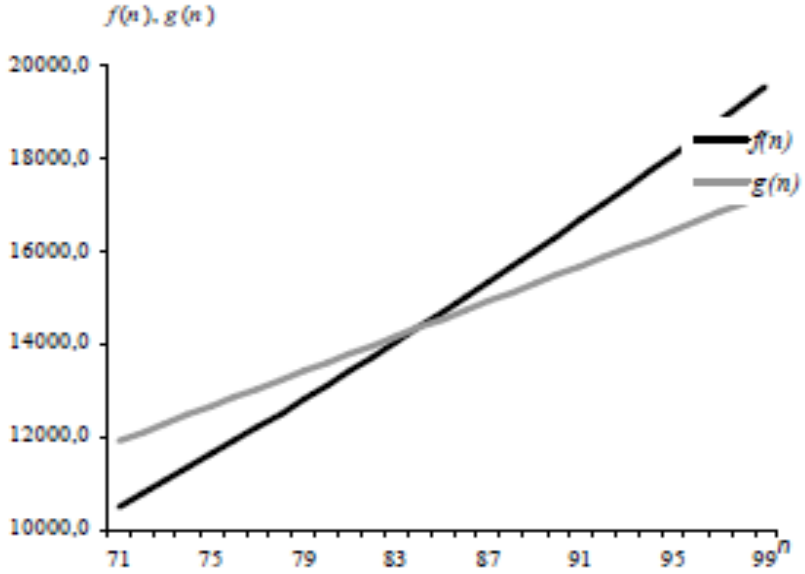


Рис. 15.2. Графіки $f(n) = 1{,}75n^2 + 2\,n\ln n + 15n + 9$ та $g(n) = 18n\,\lg n + 85n - 66$

У табл. 15.1 наведено значення функцій працеємності алгоритму сортування методом пошуку мінімуму та максимуму $f(n)$ й алгоритму сортування злиттям $g(n)$, значення міри $\pi(f(n), g(n))$ на $(n_1, n_2) = (71, 99)$ [10]:

Таблиця 15.1. Значення міри $\pi(f(n), g(n))$ алгоритмів на $(n_1, n_2) = (71, 99)$

n	$f(n)$	$g(n)$	$\pi(f(n), g(n))$
71	10501.05	11908.71	-0.1255
72	10776.84	12095.56	-0.1152
73	11056.16	12282.66	-0.1050
74	11339.00	12470.02	-0.0949
75	11625.37	12657.61	-0.0850
76	11915.27	12845.44	-0.0751
77	12208.70	13033.51	-0.0653
78	12505.65	13221.82	-0.0557
79	12806.12	13410.36	-0.0461
80	13110.12	13599.12	-0.0366
81	13417.65	13788.11	-0.0272
82	13728.70	13977.32	-0.0179
83	14043.28	14166.75	-0.0088
84	14361.38	14356.40	0.0003
85	14683.00	14546.27	0.0094
86	15008.15	14736.33	0.0183
87	15336.82	14926.61	0.0271
88	15669.01	15117.10	0.0359
89	16004.73	15307.80	0.0445
90	16343.97	15498.69	0.0531
91	16686.73	15689.79	0.0616
92	17033.01	15881.08	0.0700
93	17382.81	16072.57	0.0783
94	17736.14	16264.26	0.0865
95	18092.99	16456.13	0.0947
96	18453.36	16648.19	0.1028
97	18817.24	16840.45	0.1108
98	19184.65	17032.88	0.1187
99	19556.58	17225.50	0.1265

За виконаним аналізом: за вибраного значення $\varphi = \pi/32 \approx 0{,}0981$ асимптотично більш повільний квадратичний алгоритм переважний для сортування масивів вимірністю входу n до 73, в інтервалі від 74 до 95 обидва алгоритми можуть бути однаково застосовувані з точністю до

обраного значення φ міри; якщо *n* більше 95, алгоритм сортування злиттям є більш ефективним за функцією працеємності.

Області еквівалентної ресурсної ефективності алгоритмів. Засоби аналізу ресурсних функцій на скінченому інтервалі є складовою порівняльного аналізу ресурсної ефективності комп'ютерних алгоритмів (виявлення можливої кількості та умов існування областей ресурсної ефективності).

Області еквівалентної ресурсної ефективності – це такі інтервали вимірностей початкових даних задачі, в яких значення ресурсних функцій алгоритмів-претендентів розрізняються не більш ніж на заданий поріг φ прийнятої міри.

Так як функції $f_A(x)$ та $g_A(x)$, що обчислюються в цілочисельних точках $x = n$, є ресурсними функціями алгоритмів, то вони задовольняють умовам: $f_A(x) > 0$, $g_A(x) > 0$ за $x > 1$; $f_A(x)$ та $g_A(x)$ неперервні та двічі диференційовані на будь-якому $[n_1, n_2]$; похідні функцій $f_A(x)$ та $g_A(x)$ невід'ємні.

Тоді, у припущенні переходу до неперервного аргументу, за заданого порогу φ на інтервалі (a, b) по відношенню до функцій $f_A(x)$ та $g_A(x)$ кажуть, що **існує Θ_φ-область**, тобто $f_A(n) = \Theta_\varphi(g_A(n))$, якщо

$$\max|\{\pi(f_A(n), g_A(n))\}| < \varphi \quad \forall x = (a, b).$$

Довільна точка x_0, що належить досліджуваному інтервалу вимірностей $[n_1, n_2]$, належить до деякої **Θ_φ-області**, якщо на заданому значенні φ

$$|\{\pi(f_A(x_0), g_A(x_0))\}| < \varphi.$$

Ширина Θ_φ областей та їх кількість у досліджуваному сегменті вимірностей $[n_1, n_2]$ визначаються функціями $f_A(x)$, $g_A(x)$ та значенням порогу φ, що визначає рівнозастосовуваність алгоритмів. Наявність або відсутність на $[n_1, n_2]$ Θ_φ-області визначається близькістю значень функції $f_A(x)$ та $g_A(x)$ на цьому інтервалі за прийнятою мірою.

Для дослідження Θ_φ-областей уведемо наступні функції:

$$u(x) = f_A(x) - g_A(x),$$
$$u_1(x) = |f_A(x) - g_A(x)|,$$
$$d(\varphi, x) = \begin{cases} 1, & |\pi(f_A(x), g_A(x))| < \varphi, \\ 0, & |\pi(f_A(x), g_A(x))| \geq \varphi, \end{cases}$$

$$h(x,\varepsilon) = u(x-\varepsilon)u(x+\varepsilon),\ \varepsilon > 0\,.$$

Так як $\pi(f_A(x), g_A(x))$ неперервна, то на заданому сегменті вимірностей $[n_1, n_2]$ знайдеться така точка ξ, що $\pi(f_A(\xi), g_A(\xi))$ в цій точці дорівнює точній нижній границі значень $\pi(f_A(x), g_A(x))$ на $[n_1, n_2]$,

$$\exists\, \xi\colon \pi(f_A(\xi), g_A(\xi)) = \min\{\pi(f_A(x), g_A(x))\},\ x \in [n_1, n_2],\ \xi \in [n_1, n_2].$$

Якщо $d(\varphi, \xi) = 1$, то за заданого порогу φ на $[n_1, n_2]$ існує хоча б одна Θ_φ-область. Якщо ж $d(\varphi, \xi) = 0$, то за заданого порогу φ на $[n_1, n_2]$ Θ_φ-областей не існує та один з алгоритмів на всьому інтервалі переважніше за інший з порогом φ. Переважність визначається залежно від того, чи виконується одна з наступних рівностей: $f(n) = \Delta_\varphi(g(n))$ або $f(n) = o_\varphi(g(n))$ у будь-якій точці $[n_1, n_2]$.

Якщо $d(\varphi, \xi) = 1$, то важливим є як буде себе вести виявлена Θ_φ-область при зменшенні порогу.

При зменшенні φ можливі випадки:

- Θ_φ-область зберігається за будь-яких малих значень φ;
- Θ_φ-область перестає існувати починаючи з деякого значення φ.

Відповідну точці ξ область називають **φ-незалежною Θ_φ-областю**, якщо за $\varphi \to 0$ $\exists \varepsilon > 0, \delta > 0 : d(\varphi,x) = 1, \forall x \in [\xi - \varepsilon, \xi + \delta]$ (рис. 15.3).

Т.я. Θ_φ-область залежить вид вигляду $f_A(x)$, $g_A(x)$ та міри й не обов'язково є симетричною відносно точки ξ, то значення ε, δ вказують довжину лівої та правої частин області.

Відповідну точці ξ область називають **φ-залежною Θ_φ-областю**, якщо $\exists \varphi_0 : \forall \varphi < \varphi_0\ \exists \varepsilon > 0, \delta > 0 : d(\varphi,x) = 1, x \in [\xi - \varepsilon, \xi + \delta]$ та $\forall \varphi \geq \varphi_0$ $d(\varphi,x) = 0$ (рис. 15.3).

Твердження. На $[n_1, n_2]$ **φ-незалежна Θ_φ-область** для функцій $f_A(x)$, $g_A(x)$ (що задовольняють умовам невід'ємності, неперервності, двічі диференційованості, невід'ємності похідних) існує в точці ξ, в якій $u_1(\xi) = \min\{u_1(x)\}, x \in [n_1, n_2]$, тоді й тільки тоді, коли $u_1(\xi) = 0$ [10].

Виконання умови $u_1(\xi) = 0$ еквівалентне тому, що функції $f_A(x)$, $g_A(x)$ або перетинаються в точці ξ, або дотикаються в цій точці (що саме – визначаємо за знаком функції $h(\xi, \varepsilon)$).

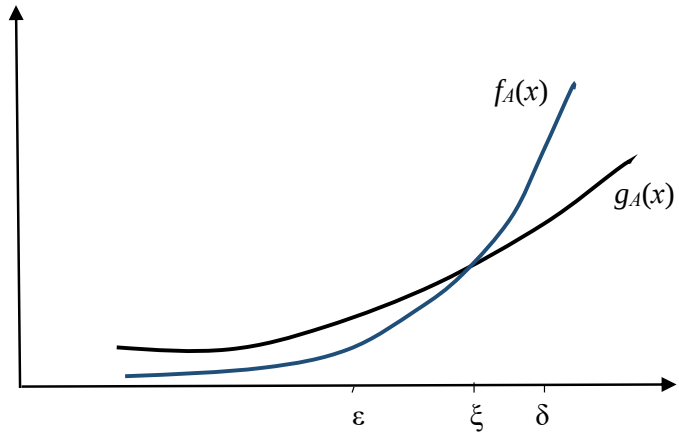


Рис. 15.3. Вибір раціонального алгоритму на заданому інтервалі

Якщо за $\varepsilon \to 0$ значення $h(\xi, \varepsilon) > 0$, функції дотикаються у точці ξ, якщо за $\varepsilon \to 0$ значення $h(\xi, \varepsilon) < 0$, то вони перетинаються, забезпечуючи в кожному випадку φ-незалежну Θ_φ-область.

Твердження. Якщо функції $f_A(x)$, $g_A(x)$ неперервні та двічі диференційовані на $[n_1, n_2]$ й друга похідна функції $u(x) = f_A(x) - g_A(x)$ не перетворюється в 0 на $[n_1, n_2]$, то функції $f_A(x)$, $g_A(x)$ мають на $[n_1, n_2]$ не більше двох точок перетину або одної точки дотику [10].

Практичне застосування результату: для деяких ресурсних функцій алгоритмів на інтервалі $[n_1, n_2]$ можливе існування двох φ-незалежних Θ_φ-областей за визначеного значення порогу φ, що необхідно враховувати при проведенні порівняльного аналізу алгоритмів. Одна з можливих ситуацій розбиття інтервалу $[n_1, n_2]$ на області еквівалентної ресурсної ефективності й області переваги наведена на рис. 15.4.

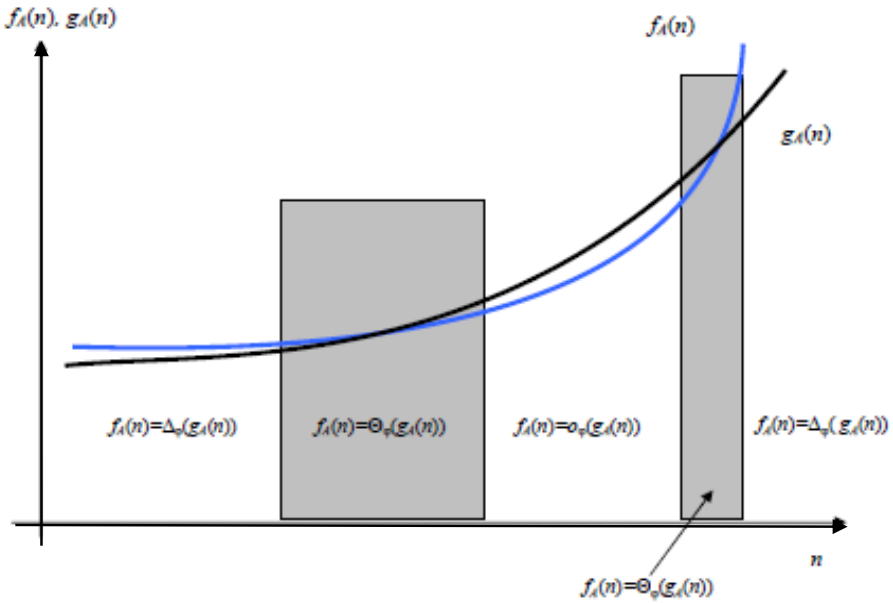


Рис. 15.4. Вибір алгоритму за двох φ-незалежних Θ_φ –областей [10]

Лекція 16. NP-повнота. Привідність. NP-складність задач

Як відомо, для рішення NP-повних задач до теперішнього часу не розроблено алгоритмів з поліноміальним часом роботи, але й не доведено, що для якої-небудь з них такого алгоритму не існує. Крім того, можемо навести приклади задач, що дуже «близькі» до задач, для яких існують алгоритми з поліноміальним часом роботи, але вони є NP-повними. Наприклад, задачі пошуку самих коротких та самих довгих простих шляхів у орієнтованому графі (найкоротші шляхи в графі G = (V, E) можна знайти за час O(VE), а пошук самого довгого шляху між двома вершинами не є такою задачею – задача визначення чи містить граф G простий шлях, кількість ребер в якому не менша заданого числа, є NP-повною задачею).

Існує також клас NP, який складається із задач, що можуть бути перевірені за поліноміальний час. Тобто, якщо деяким чином отриманий

«сертифікат» рішення, то за час, що поліноміально залежить від розміру вхідних даних задачі, можна перевірити коректність такого рішення. Наприклад, для задачі про гамільтоновий цикл із заданим орієнтованим графом G = (V, E) сертифікат мав би вигляд послідовності $\langle v_1, v_2, ..., v_{|V|} \rangle$ з $|V|$ вершин. Впродовж поліноміального часу легко перевірити, чи $(v_i, v_{i+1}) \in E$ для $i = 1,2,...,|V|-1$ та чи $(v_{|V|}, v_i) \in E$. Будь-яка задача класу P належить класу NP (навіть не потрібно мати сертифікату).

Можна сказати, що задача належить класу NPC (NP-повних), якщо вона належить класу NP та є такою ж «складною, як і будь-яка з класу NP». Якщо встановлено, що задача є NP-повною, то доцільною є розробка наближеного алгоритму або рішення простого часткового випадку. Щоб показати, що задача є NP-повною, робиться твердження про те, наскільки ця задача є складною (на відміну від того, як звичайно намагаємося довести існування ефективного алгоритму). Для цього спершу необхідно задачу представити як задачу прийняття рішення, в якій відповідь може бути позитивною або негативною. Наприклад, у задачі пошуку найкоротшого шляху маємо неорієнтований граф G, деякі вершини *u, v*. Необхідно знайти шлях від вершини *u* до вершини *v* з найменшою кількістю ребер (що є задачею оптимізації). Так як будь-яку задачу оптимізації можна представити як задачу прийняття рішень (обмеживши деяким чином оптимізоване значення), то задачі пошуку найкоротшого шляху буде відповідати задача прийняття рішення про існування шляху, що складається не більше ніж з *k* ребер (чи існує для заданого графу G та заданого числа *k* шлях з вершини *u* до вершини *v*, що складається не більше ніж з *k* ребер).

Для дослідження наскільки задана оптимізаційна задача є складною можна використати відповідну їй задачу прийняття рішення, так як вона *не є більш складною* задачею. Тобто, якщо покажемо складність задачі прийняття рішення, то відповідна задача оптимізації є також складною. Крім того, щоб показати належність до класу NP-повних задач, можна й у випадку двох задач прийняття рішення використовувати, що одна задача не складніша або не легша за іншу. Наприклад, маємо деяку задачу прийняття рішення A, а саме екземпляр α задачі A. Нехай існує інша задача прийняття рішення B, для якої наперед відомо, як можна вирішити її за поліноміальний час, а також є процедура, яку

називають **алгоритмом приведення**, що перетворює будь-який екземпляр α задачі А у деякий екземпляр β задачі В (процедура виконує перетворення за поліноміальний час, відповіді задач будуть однаковими – в екземплярі α задачі А відповідь «так» тоді й тільки тоді, коли в екземплярі β задачі В теж відповідь «так»). Причому алгоритм приведення з поліноміальним часом дає можливість отримати рішення задачі А виконавши: перетворення заданого екземпляра α задачі А в екземпляр β задачі В за поліноміальний час; запуск алгоритму, що вирішує екземпляр β задачі В за поліноміальний час, надання відповіді для екземпляра β задачі В як відповіді для екземпляра α задачі А. Щоб показати належність задачі А до класу NP-повних задач з використанням алгоритму приведення з поліноміальним часом, для задачі прийняття рішення В наперед має бути відомим, що для неї не існує алгоритму з поліноміальним часом роботи. Тоді «від супротивного» легко отримати, що для рішення задачі А не існує алгоритмів з поліноміальним часом роботи [1, 11].

Використаємо поняття **абстрактної задачі** Q, як бінарне відношення між множиною екземплярів задач I та множиною рішень задач S. Наприклад, екземпляр задачі про пошук найкоротшого шляху складається з таких елементів: графу та двох вершин. Рішенням є послідовність вершин графу. Сама задача представляє собою відношення, що співставляє кожному екземпляру графу та двом його вершинам найкоротший шлях по графу, який поєднує ці дві вершини (і такий шлях може бути не єдиним). Для задачі прийняття рішення абстрактну задачу розуміємо як функцію, що відображає екземпляр множини I на множину рішень $\{0, 1\}$.

Комп'ютерний алгоритм, за допомогою якого вирішується деяка абстрактна задача прийняття рішення, приймає в якості вхідних даних закодований екземпляр задачі. Задачу, множина екземплярів якої є множиною бінарних рядків, називають **конкретною**. Кажуть, що алгоритм розв'язує конкретну задачу за час $O(\mathrm{T}(n))$, якщо для заданого екземпляра задачі i довжиною $n = |i|$ з його допомогою можна одержати рішення за час $O(\mathrm{T}(n))$. Визначимо **клас складності P** як множину конкретних задач прийняття рішень, що вирішуються за поліноміальний час.

За допомогою кодування абстрактні задачі можна відображати на конкретні, а саме задану абстрактну задачу прийняття рішення Q, що

відображає множину екземплярів I на множину {0, 1}, за допомогою кодування $e : I \to \{0,1\}^*$ (множина всіх рядків, що складаються з символів множини {0, 1}) можна звести до пов'язаної з нею конкретної задачі прийняття рішення e(Q). Якщо $Q(i) \in \{0,1\}$ є рішенням екземпляра абстрактної задачі $i \in I$, то воно ж є рішенням екземпляру конкретної задачі $e(i) \in \{0,1\}^*$. Тобто конкретна задача має ті ж рішення, що й абстрактна, якщо її екземпляри у вигляді бінарних рядків є закодованими екземплярами абстрактної задачі. Але в залежності від способу кодування алгоритм може виконуватися за поліноміальний час або за час більший за поліноміальний. Тому кодування абстрактної задачі є дуже важливим питанням й неможливо казати про рішення абстрактної задачі без докладного опису кодування. Але саме кодування задачі мало впливатиме на те, чи вирішувана задача за поліноміальний час.

Функція $f : \{0,1\}^* \to \{0,1\}^*$ **обчислювана за поліноміальний час**, якщо існує алгоритм A з поліноміальним часом роботи, який для довільних даних $x \in \{0,1\}^*$ повертає вихідні дані $f(x)$. Для деякої множини I екземплярів задачі два кодування e_1, e_2 називають **поліноміально зв'язаними**, якщо існують такі дві обчислювані за поліноміальний час функції f_{12}, f_{21}, що для будь-якого екземпляру $i \in I$ виконується: $f_{12}(e_1(i)) = e_2(i)$, $f_{21}(e_2(i)) = e_1(i)$ (тобто закодовану величину $e_2(i)$ можна обчислити на основі закодованої величини $e_1(i)$ за допомогою алгоритму з поліноміальним часом роботи та навпаки). Якщо ж два кодування e_1, e_2 абстрактної задачі є поліноміально зв'язаними, то вирішуваність задачі за поліноміальний час не залежить від використовуваного кодування. А саме (якщо тільки кодування не унарне):

Лема [1]. Нехай Q є деякою абстрактною задачею прийняття рішення, визначеною на множині екземплярів I, e_1, e_2 – поліноміально зв'язані кодування множини I. Тоді $e_1(Q) \in P$ тоді і тільки тоді, коли $e_2(Q) \in P$.

Схема формальних мов дозволяє виразити взаємовідношення між задачами прийняття рішення та алгоритмами, що їх вирішують.

Мовою L, визначеною над абеткою Σ (як скінченою множиною символів), є довільна множина рядків, що складаються з символів множини Σ. Мову всіх рядків, заданих над множиною Σ, позначимо Σ^*. Будь-яка мова L над множиною Σ є підмножиною множини Σ^*. Над мовами

визначають операції об'єднання, перетину, доповнення ($\overline{L} = \Sigma^* - L$), конкатенації, замикання. Множина екземплярів будь-якої задачі прийняття рішення Q є множиною Σ^*, де $\Sigma = \{0, 1\}$. Так як множина Q повністю характеризується тими екземплярами задачі, що дають відповідь 1, то множину можна розглядати як мову L над множиною $\Sigma = \{0, 1\}$ ($L = \{x \in \Sigma^* : Q(x) = 1\}$). **Алгоритм А приймає** рядок $x \in \{0,1\}^*$, якщо для заданих вхідних даних x вихід алгоритму A(x) дорівнює 1. **Мова, що приймається алгоритмом** A, є множиною рядків L = { $x \in \{0,1\}^*$: A(x) = 1}, тобто множиною рядків, що приймається алгоритмом. Алгоритм А **відкидає** рядок x, якщо A(x) = 0. Якщо мова L приймається алгоритмом А, то цей алгоритм не обов'язково відкидає рядок $x \notin$ L, який подається в якості вхідних даних. Мова L **розпізнається** алгоритмом А, якщо кожен бінарний рядок цієї мови приймається алгоритмом А, а кожен бінарний рядок, що не належить мові L, відкидається цим алгоритмом. Мова L **приймається за поліноміальний час** алгоритмом А, якщо вона приймається алгоритмом та існує така стала k, що для довільного n-символьного рядку $x \in$ L алгоритм А приймає рядок x за час $O(n^k)$. Мова L **розпізнається за поліноміальний час** алгоритмом А, якщо існує така стала k, що для довільного n-символьного рядку $x \in \{0,1\}^*$ алгоритм за час $O(n^k)$ вірно зрозуміє, чи належить рядок x мові L [1].

Використовуючи теорію формальних мов визначення класу складності P приймає наступний вигляд: P = {L $\subseteq \{0,1\}^*$: існує алгоритм А, що вирішує мову L за поліноміальний час}. Тому фактично P є також класом мов, що можуть бути прийняті за поліноміальний час.

Розглянемо питання перевірки належності мові за поліноміальний час на прикладі задачі про пошук гамільтонових циклів [1], тобто циклів неорієнтованого графу G = (V, E), що є простими та містять усі вершини множини V. Задачу можна визначити як формальну мову HAM-CYCLE ={$\langle G \rangle$: G є гамільтоновим графом}. Алгоритм міг би розпізнати мову HAM-CYCLE наступним чином. Для заданого екземпляру задачі $\langle G \rangle$ можна узяти алгоритм прийняття рішення, який би спочатку формував список всіх перестановок вершин графу G, а далі перевіряв би кожну перестановку на наявність гамільтонового шляху.

Час роботи такого алгоритму може бути: $\Omega(m!) = \Omega(\sqrt{n}!) = \Omega(2^{\sqrt{n}})$ ($m!$ можливих перестановок вершин, кількість яких $m \in \Omega(\sqrt{n})$, де $n = |\langle G \rangle|$ – довжина коду графу G. Отримана величина не є величиною порядку $O(n^k)$ ні за якої сталої k, тобто час роботи подібного алгоритму не є поліноміальною функцією. Але перевірити, чи є деяка послідовність вершин гамільтоновим циклом, можна за поліноміальний час, а саме за час $O(n^2)$, де n – довжина коду графу G.

Визначимо **алгоритм верифікації** як алгоритм A з двома аргументами, один з яких є звичайним рядком входу x, а другий – бінарним рядком y (називають *сертифікатом*). Двохаргументний алгоритм A **верифікує** вхідний рядок x, якщо існує сертифікат y, такий, що $A(x, y) = 1$.

Мова, що верифікована алгоритмом верифікації A, є множиною $L = \{ x \in \{0,1\}^* :$ існує $y \in \{0,1\}^*$, такий, що $A(x, y) = 1\}$.

Алгоритм A верифікує мову L, якщо для будь-якого рядку $x \in L$ існує сертифікат y, що дозволяє довести за допомогою алгоритму A, що $x \in L$. Для будь-якого рядку $x \notin L$ не має існувати сертифікату, який доводить, що $x \notin L$ (наприклад, якщо граф не є гамільтоновим, то не існує списку вершин, що дозволяє обдурити алгоритм верифікації й встановити, шо граф є гамільтоновим, так як алгоритм верифікації виконує ретельну перевірку запропонованого циклу).

Отже, **клас складності NP** – це *клас мов, які можна верифікувати за допомогою алгоритму з поліноміальним часом роботи*. Мова L належить класу NP тоді і тільки тоді, коли існує алгоритм A з двома вхідними параметрами та поліноміальним часом роботи, стала c, така, що $L=\{ x \in \{0,1\}^* :$ існує сертифікат y з $|y| = O(|x|^c)$, такий, що $A(x,y) = 1\}$.

Тоді кажуть, що **алгоритм A верифікує мову** L за поліноміальний час. Отже, мова HAM-CYCLE $\in$ NP. Крім того, мова HAM-CYCLE є NP-повною задачею.

Привідність. Задачу Q можна *звести* до іншої задачі Q', якщо будь-який екземпляр задачі Q перефразується в екземпляр задачі Q', рішення якого дозволяє одержати рішення відповідного екземпляру задачі Q. Якщо задача Q зводиться до іншої задачі Q', то розв'язати задачу Q у деякому смислі «*не складніше*», ніж задачу Q'. Кажуть, що **мова L_1 привідна за поліноміальний час до мови L_2** ($L_1 \leq_P L_2$), якщо

існує обчислювана за поліноміальний час функція $f:\{0,1\}^* \to \{0,1\}^*$ така, що для всіх $x \in \{0,1\}^*$ $x \in L_1$ тоді і тільки тоді, коли $f(x) \in L_2$. Функцію f називають **функцією приведення**, а алгоритм з поліноміальним часом роботи, що обчислює функцію f – **алгоритмом приведення**. Отже функція приведення відображає будь-який екземпляр x задачі прийняття рішення, представлений мовою L_1 на екземпляр $f(x)$ задачі, представлений мовою L_2 [1].

Приведення є важливим інструментом у доведенні, що різні мови належать класу P, та формальним засобом, який дозволяє показати, що одна задача за складністю є такою ж, як і інша, хоча б з точністю до поліноміально-часового множника (тобто якщо $L_1 \leq_P L_2$, то складність мови L_1 перевищує складність мови L_2 не більше ніж на поліноміальний множник).

Лема. Якщо $L_1, L_2 \subseteq \{0,1\}^*$ є мовами, такими, що $L_1 \leq_P L_2$, то з $L_2 \in P$ випливає, що $L_1 \in P$ [1].

Мова $L \subseteq \{0,1\}^*$ є **NP-повною**, якщо $L \in NP$ та $L' \leq_P L$ для кожного $L' \in NP$.

Якщо мова L задовольняє другій частині визначення, але не обов'язково $L \in NP$, то мова L є **NP-складною.**

Теорема. Якщо деяка NP-повна задача вирішувана за поліноміальний час, то P = NP (твердження еквівалентне до: якщо яка небудь задача з класу NP не вирішується за поліноміальний час, то й жодна з NP-повних задач не вирішується за поліноміальний час) [1].

Доведення. Нехай $L \in P$ та $L \in NPC$. Тоді для довільної мови $L' \in NP$ з визначення NP-повноти маємо $L' \leq_P L$ для кожної $L' \in NP$. За попередньою лемою отримаємо $L' \in P$.

Використаємо поняття булевих комбінаційних схем для доведення NP-повноти задачі по виконуваність схем. Булеві комбінаційні схеми складаються з булевих комбінаційних елементів, що поєднані між собою проводами, які можуть з'єднувати вихід одного елементу із входом іншого. До одного проводу не можна підключати більше одного виводу елементу, проте сигнал з нього може поступати одночасно на входи різних елементів. Булевий комбінаційний елемент (комбінаційний логічний елемент) – це довільний елемент мережі, що має фіксовану кількість булевих входів та виходів та виконує деяку визначену

функцію. Комбінаційні логічні елементи, використовувані в задачі про виконуваність схем, обчислюють просту булеву функцію (їх називають логічними елементами). Використовуються 3 основних логічних елемента: NOT, AND, OR. На елемент NOT (¬) поступає одна бінарна вхідна величина *x*, значення якої є 0 або 1, видає він бінарну вихідну величину z, значення якої протилежне значенню вхідної величини. На елементи AND (∧), OR (∨) поступає дві бінарні вхідні величини *x* та *y*, видають вони одну бінарну вихідну величину z. Роботу кожного логічного та кожного комбінаційного елемента можна описати таблицею істинності. Кількість елементів, для яких провід надає вхідні дані називають коефіцієнтом розгалуження. Провід, до якого не під'єднані виходи жодного елементу, називають входом схеми, що отримує вхідні дані від зовнішніх джерел. Провід, до якого не під'єднаний вхід жодного елементу, називають виходом схеми, що видає результати роботи схеми. Логічна комбінаційна схема не має циклів, тобто, якщо схемі співставити орієнтований граф з вершинами у кожному комбінаційному елементі та *k* орієнтованими ребрами для кожного проводу (з коефіцієнтом розгалуження *k*, ребру відповідає провід, що поєднує вихід одного елементу із входом іншого), то такий граф буде ациклічним. Розмір логічної комбінаційної схеми визначають як суму кількості комбінаційних елементів та числа проводів у схемі.

Розглянемо набори значень булевих змінних, що відповідають входам системи. Логічна комбінаційна **схема виконувана**, якщо вона має виконуючий набір (набір значень, за подачі якого на вхід схеми, її вихідне значення дорівнює 1).

Задача про виконуваність схеми: чи виконується задана логічна комбінаційна схема, що складається з елементів NOT, AND, OR.

Для вирішення такого питання необхідно виконати узгодження щодо стандартного коду для подібних схем (наприклад код, що використовується при представленні графів, що відображає кожну задану схему C на бінарний рядок $\langle C \rangle$, довжина якого описується не більше ніж поліноміальною функцією від розміру схеми). Визначимо формальну мову CIRCUIT-SAT = { $\langle C \rangle$: C є виконуваною булевою схемою}.

Звичайно, щоб перевірити, чи виконується схема, можна спробувати перебрати всі можливі комбінації вхідних значень. Але, якщо в схемі *k* входів, то матимемо 2^k комбінацій. Якщо розмір схеми C виражається

поліноміальною функцією від k, то перевірка всіх комбінацій займатиме час $\Omega(2^k)$, що за швидкістю зростання переважає поліноміальну функцію.

Лема. Задача про виконуваність схем CIRCUIT-SAT належить класу NP [1].

Доведення. Сформулюємо алгоритм А з двома вхідними параметрами та поліноміальним часом роботи, що здатний перевірити рішення задачі CIRCUIT-SAT. Одним з вхідних параметрів алгоритму А є логічна комбінаційна схема С (її стандартний код). Ще одним – сертифікат, який є булевими значеннями у проводах схеми. Алгоритм А: для кожного логічного елементу схеми перевіряється що надане сертифікатом значення на вихідному проводі вірно обчислюється як функція значень на вхідних проводах. Якщо на вихід схеми подається значення 1, то алгоритм також видає 1, так як величини, що поступили на вхід схеми С, є виконуючим набором. Інакше алгоритм виводить 0.

Для довільної виконуваної схеми С, що виступає як вхідний параметр алгоритму А, існує сертифікат, довжина якого описується поліноміальною функцією від розміру схеми С, та який приводить до подачі на виході алгоритму А значення 1. Для довільної невиконуваної схеми, що виступає як вхідний параметр алгоритму А, жодний сертифікат не може впевнити алгоритм в тому, що ця схема є виконуваною. Алгоритм А виконується за поліноміальний час, тому CIRCUIT-SAT $\in$ NP.

Комп’ютерна програма зберігається в пам’яті комп’ютера як послідовність інструкцій. Типова інструкція містить код операції, яку потрібно виконати, адреси операндів в пам’яті та адресу, куди буде розміщений результат. Лічильник команд слідкує за тим, яка інструкція має виконуватися наступною. Коли отримали наступну команду, то покази лічильника автоматично зростають на 1. Це дає можливість послідовного виконання інструкцій, але за результатом виконання певних інструкцій у лічильник команд може бути записане деяке значення, що змінює звичайний послідовний порядок виконування. У будь-який момент виконання програми стан обчислень у загальному можна представити вмістом пам’яті комп’ютера (області пам’яті, що складається з програми, лічильника команд, робочої області та інших бітів щодо стану програми). Виконання команд можна розглядати як відображення однієї конфігурації пам’яті (як деякого окремого стану пам’яті комп’ю

тера) на іншу. Апаратне забезпечення, що здійсняє таке відображення, можна реалізувати як логічну комбінаційну схему, яку позначимо M.

Лема. Задача про виконуваність схем CIRCUIT-SAT є NP-складною [1].

Доведення. Нехай L є мовою класу NP. Опишемо алгоритм F з поліноміальним часом роботи, що обчислює функцію приведення *f*, яка відображає кожний бінарний рядок x на схему $C = f(x)$, таку, що $x \in L$ тоді і тільки тоді, коли $C \in$ CIRCUIT-SAT.

Так як $L \in NP$, то має існувати алгоритм A, що верифікує мову L за поліноміальний час. В алгоритмі F, який побудуємо далі, для обчислення функції приведення *f* використовуємо алгоритм A з двома вхідними параметрами. Нехай $T(n)$ – час роботи алгоритму A у найгіршому випадку для вхідного рядку довжиною n, $T(n) = O(n^k)$, де стала $k \geq 1$, довжина сертифікату дорівнює $O(n^k)$. Представимо виконання алгоритму A як послідовність конфігурацій, де кожну конфігурацію можна розбити на частини: програма алгоритму A, лічильник команд комп'ютера, регістри процесора, вхідні дані x, сертифікат y, робоча пам'ять. Починаючи з початкової конфігурації c_0, кожна конфігурація c_i за допомогою комбінаційної схеми M, апаратно реалізованої у комп'ютері, відображається на наступну конфігурацію c_{i+1}. Вихідне значення алгоритму A (0 або 1) по завершенню виконання алгоритму A записується у деяку спеціально призначену комірку робочої пам'яті, причому після зупинення алгоритму A значення не змінюється. Тому, якщо виконання алгоритму відбувається не більше ніж за $T(n)$ кроків, результат його роботи зберігається у деякому біті $c_{T(n)}$.

Алгоритм приведення F будує єдину комбінаційну схему, що обчислює всі конфігурації, що одержуються із заданої вхідної конфігурації. Необхідно звести усі $T(n)$ копій схеми M. Вихідні дані i-ої схеми, що видає конфігурацію c_i, подаються на вхід i+1-ої схеми. Тому ці конфігурації не зберігаються, а просто передаються через проводи, що поєднують копії схеми M.

Алгоритм приведення F для заданих вхідних даних x має обчислити схему $C = f(x)$, яка є виконуваною тоді і тільки тоді, коли існує сертифікат y, такий, що $A(x, y) = 1$. Коли алгоритм F отримує вхідне значення x, то спочатку обчислює $n = |x|$ та будує комбінаційну схему C', що складається з $T(n)$ копій схеми M. На вхід схеми C' подається

початкова конфігурація, що відповідає обчисленню A(x, y), а виходом схеми є конфігурація $c_{T(n)}$.

Схема C = $f(x)$, створена алгоритмом F, отримується деякою модифікацією схеми C'. Входи схеми C', що відповідають програмі алгоритму A, початковому значенню лічильника команд, вхідній величині x, початковому стану пам'яті, поєднуються проводами із вказаними величинами (відомими). Тому входи схеми, що залишилися, відповідають сертифікату y. Всі виходи схеми ігноруються за виключенням одного біта конфігурації $c_{T(n)}$, який відповідає вихідному значенню алгоритму A. Вказана схема C для довільного вхідного параметра y довжиною O(n^k) обчислює C(y) = A(x, y). Алгоритм приведення F, на вхід якого подається рядок x, обчислює вказану схему C й видає її.

Покажемо, що алгоритм F вірно обчислює функцію приведення f (схема C виконувана тоді й тільки тоді, коли існує сертифікат y, такий, що A(x, y) = 1) й його робота завершується за поліноміальний час. Припустимо, що існує сертифікат y довжиною O(n^k), такий, що A(x, y) = 1. Тоді, якщо на вхід схеми C подаються біти сертифікату y, на виході схеми отримується значення C(y) = A(x, y) = 1. Тому, якщо сертифікат існує, то схема C виконувана. Нехай схема C виконувана. Тоді існує таке вхідне значення y, що C(y) = 1, а тому A(x, y) = 1. Отже, алгоритм F вірно обчислює функцію приведення f. Кількість бітів, необхідна для представлення конфігурації, поліноміально залежить від n. Об'єм програми алгоритму A фіксований і не залежить від довжини вхідного параметра x. Довжина вхідного параметра x дорівнює n, а довжина сертифікату y - O(n^k). Так як робота алгоритму складається не більше ніж з O(n^k) кроків, то об'єм потрібної йому пам'яті також виражається поліноміальною функцією від n.

Розмір комбінаційної схеми M, що реалізує апаратну частину комп'ютера, виражається поліноміальною функцією від довжини конфігурації, яка (конфігурація) є поліноміальною від O(n^k). Схема C складається не більше ніж з t = O(n^k) копій M, тому її розмір поліноміально залежить від n. Схему C для вхідного параметра x можна скласти за поліноміальний час з допомогою алгоритму приведення F, т.я. кожен етап побудови триває поліноміальний час.

Отже, мова CIRCUIT-SAT не є простішою за будь-яку мову класу NP, а так як вона ще й відноситься до класу NP, то вона є NP-повною.

Теорема. Задача про виконуваність схем є NP-повною.

Доведення NP-повноти для задачі про виконуваність схем базувалося на доведенні приведення кожної мови з класу NP до заданої мови. Покажемо, що можливо довести NP-повноту мови базуючись лише на приведенні *деякої* мови з класу NP.

Лема. Якщо мова L така, що L' $\leq_P$ L для деякого L'$\in$NPC, то L є NP-складною. Якщо, крім того, L$\in$NP, то L$\in$NPC (NP-повною) [1].

Доведення. Так як мова L'$\in$NPC, то для всіх L''$\in$NP маємо L'' $\leq_P$ L'. За умовою L' $\leq_P$ L. Тому за транзитивністю маємо: L'' $\leq_P$ L, а тому задача L є NP-складною. Якщо ж L$\in$NP, то L$\in$NPC.

Тобто, якщо мову L', що є NP-повною, можна звести до мови L, то значить до цієї мови можна звести будь-яку мову класу NP. Це дає метод доведення NP-повноти мови L, який має такі *етапи* (перший етап – доведення, що L$\in$NP, усі інші – що L є NP-складною):

- доведення, що L$\in$NP;
- вибір мови L', про яку відомо, що вона є NP-повною;
- опис алгоритму, що обчислює функцію приведення *f*, яка відображає кожен екземпляр $x \in \{0,1\}^*$ мови L' на екземпляр $f(x) \in$ L;
- доведення, що для функції приведення *f* співвідношення $x \in L'$ виконується тоді і тільки тоді, коли $f(x) \in$ L для всіх $x \in \{0,1\}^*$;
- доведення, що час роботи алгоритму, що обчислює функцію *f*, є поліноміальним.

Виконуваність формули: чи виконується задана формула, що складається з множини елементів. Задача формулюється в термінах мови SAT. Екземпляр мови SAT – це булева формула φ, що складається з: n булевих змінних $x_1, x_2, \ldots, x_n$; m булевих поєднуючих елементів (NOT (¬), AND (∧), OR (∨), імплікації (→), еквівалентності (↔)); дужок). Логічна формула φ кодується рядком, довжина якого виражається поліноміальною функцією від $n + m$. Набором значень логічної формули φ є множина значень змінних цієї формули, виконуючим набором – такий набір значень, за якого результат обчислення формули дорівнює 1. Формула, для якої існує виконуючий набор, є виконуваною.

В термінах формальних мов задача має вигляд:

SAT = { $\langle\varphi\rangle$: φ – виконувана булева формула}.

Простий прямолінійний алгоритм, що дозволяє визначити чи є виконуваною довільна булева формула, не вкладається в поліноміальний час (у формулі φ з n змінними є загалом 2^n можливих варіантів присвоєння, якщо довжина $\langle\varphi\rangle$ виражається поліноміальною функцією від n, то для перевірки кожного варіанта присвоєння потрібен час $\Omega(2^n)$).

Теорема. Задача про виконуваність булевих формул є NP-повною [1].

Доведення. Покажемо, що SAT ∈ NP. Для цього покажемо, що сертифікат, який складається з виконуючого набору вхідної формули φ, можна верифікувати за поліноміальний час. В алгоритмі верифікації кожна змінна, що міститься у формулі, замінюється відповідним значенням, й обчислюється отриманий вираз. Це виконується за поліноміальний час. Якщо в результаті отримаємо 1, то формула виконувана. Отже, SAT ∈ NP.

Покажемо, що SAT є NP-складною. Для цього покажемо, що CIRCUIT-SAT $\leq_P$ SAT. Тобто будь-який екземпляр задачі про виконуваність схеми можна за поліноміальний час звести до екземпляра задачі про виконуваність формули. За індукцією будь-яку логічну комбінаційну схему можна виразити у вигляді булевої функції (достатньо розглянути елемент, що видає вихідне значення схеми й виразити кожне вхідне значення цього елемента у вигляді формул). Але використовуючи такий метод для приведення в поліноміальний час не вкластися. Розглянемо інший метод приведення. Кожному проводу x_i схеми C співставляєься однойменна змінна формули φ. Тоді дію кожного елементу можна виразити у вигляді невеликої формули (підвиразу), що містить змінні, які відповідають проводам, приєднаним до цього елементу. Формула φ, яка є результатом виконання алгоритму приведення, отримується шляхом кон'юнкції (тобто елементу AND) вихідної змінної схеми з виразом, який є кон'юнкцією узятих у дужки підвиразів, що описують дію кожного елементу. Таку формулу φ для заданої схеми C можна отримати за поліноміальний час. Схема C виконувана тоді і тільки тоді, коли виконувана формула φ, т.я., якщо у схеми C є виконуючий набір, то по кожному проводу схеми передається деяке визначене значення, а на виході отримується 1. Набір значень, що передається по проводах схеми, у формулі φ приведе до того, що значення

кожного підвиразу у дужках у формулі φ буде 1, а тому кон'юнкція всіх підвиразів буде рівною 1. Аналогічно, якщо існує набір даних, для якого значення формули φ є 1, то схема C виконувана. Тому маємо CIRCUIT-SAT $\leq_P$ SAT.

Отже, задача про виконуваність формул є NP-повною. Існує багато задач, які можна звести до задачі про виконуваність формул, щоб довести їх NP-повноту. Але алгоритм приведення має працювати з довільними вхідними формулами, що може привести до величезної кількості випадків, які необхідно розглянути. Щоб зменшити їх кількість, бажано виконувати приведення за допомогою спрощеної версії мови булевих формул, в якості якої береться 3-кон'юнктивна нормальна форма 3-CNF-SAT.
3-CNF-виконуваність: чи виконувана задана формула φ, що належить класу 3-CNF.
Булева формула є приведеною до кон'юнктивної нормальної форми (CNF), якщо вона має вигляд кон'юнкції виразів, кожен з яких є диз'юнкцією одного або декількох літералів (літералом називають змінну, що входить до булевої формули, або заперечення змінної). Булева формула виражена у 3-CNF, якщо у кожному підвиразі у дужках міститься три різних літерала (наприклад, $(x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee \neg x_4 \vee x_5)$).
Теорема. Задача про виконуваність булевих формул у 3- кон'юнктивній нормальній формі є NP-повною [1].
Теорему приймемо без доведення.

Лекція 17. NP-складні задачі

Покажемо NP-складність низки задач, використовуючи приведення до мов, повноту яких доведено [1].
Задача про кліку. Кліка неорієнтованого графу G = (V, E) – це підмножина $V' \subseteq V$ вершин, кожна пара в якій зв'язана ребром з множини E (це повний підграф графу G). Розмір кліки – кількість вершин, що містяться в цьому підграфі. Задача про кліку є оптимізаційною задачею: потрібно знайти кліку максимального розміру, що міститься в

заданому графі. Відповідна задача прийняття рішення: чи міститься у графі кліка заданого розміру *k*.

Формальне визначення мови має вигляд: CLIQUE = { $\langle G, k \rangle$: G – граф, що містить кліку розміром *k*}.

Простий алгоритм визначення, чи містить граф G = (V, E) з $|V|$ вершинами кліку розміром *k*, складається з перечислення всіх *k*-елементних підмножин множини V та перевірки, чи утворює кожна з них кліку. Час роботи такого алгоритму є $\Omega(k^2 \binom{|V|}{k})$ та є поліноміальним для *k* сталої. Але величина *k* може досягати значень близьких до $|V|/2$, а тоді час роботи алгоритму перевищуватиме поліноміальний.

Теорема. Задача про кліку є NP-повною [1].

Доведення. Покажемо, що CLIQUE $\in$ NP. Для цього для заданого графу G = (V, E) візьмемо множину вершин кліки $V' \subseteq V$ як сертифікат для графу G. Перевірити чи є множина вершин V' клікою можна за поліноміальний час (перевірити для кожної пари вершин з V' належність ребра, що їх поєднує, множині E).

Покажемо, що 3-CNF-SAT $\leq_P$ CLIQUE, щоб показати NP-складність задачі про кліку. Алгоритм приведення починається з екземпляру задачі 3-CNF-SAT. Нехай $\varphi = c_1 \wedge c_2 \wedge \ldots \wedge c_k$ – булева формула з класу 3-CNF з *k* підвиразами. Кожен підвираз містить три різних літерала l_1^r, l_2^r, l_3^r, *r* = 1, 2,…, *k*. Побудуємо такий граф G, щоб формула φ була виконуваною тоді і тільки тоді, коли граф G містить кріку розміру *k*. Для кожного підвиразу $c_r = (l_1^r, l_2^r, l_3^r)$ у формулі φ містимо вершини v_1^r, v_2^r, v_3^r у множині V. Вершини v_i^r та v_j^s з'єднуються ребром, якщо ці вершини знаходяться у різних підвиразах ($r \neq s$) та їх літерали є сумісними (тобто l_i^r не є запереченням l_j^s). Такий граф можна побудувати для формули φ за поліноміальний час. Покажемо, що таке перетворення формули φ у граф G є приведенням. Припустимо, що для формули φ є виконуючий набір. Тоді підвираз $c_r = (l_1^r, l_2^r, l_3^r)$ містить не менше одного літерала l_i^r, значення якого дорівнює 1, й кожен такий літерал відповідає вершині v_i^r. В результаті витягання саме такого літералу з кожного підвиразу отримується множина V', що складається з *k*

вершин. Стверджуємо, що V' – кліка. Для будь-яких двох вершин v_i^r та v_j^s з V', $r \neq s$, обидва відповідних літерала l_i^r та l_j^s, за даного виконуючого набору дорівнюють 1, тому вони не можуть бути запереченням один одного. Тому ребро (v_i^r, v_j^s) належить множині E. І навпаки, нехай граф G містить кліку V' розміром *k*. Жодне з її ребер не поєднує вершини однієї і тієї ж трійки, тому кліка V' містить тільки по одній вершині з кожної трійки. Кожному літералу l_i^r, такому, що $v_i^r \in V'$, можна присвоїти значення 1 й не боятися, що воно буде надане як літералу, так і його доповненню, т.я. граф G не містить ребер, що поєднують протиречиві літерали. Кожен підвираз є виконуваним, тому виконувана і формула φ.

При доведенні теореми довільний екземпляр задачі 3-CNF-SAT зведений до екземпляру задачі CLIQUE, що має визначену структуру. Але в дійсності, хоч NP-складність задачі CLIQUE показана для окремого випадку, цього доведення достатньо, щоб зробити висновок про NP-складність задачі для графу загального вигляду (з наявності алгоритму з поліноміальним часом роботи, що вирішує задачу CLIQUE з графами загального вигляду, випливає існування такого алгоритму рішення цієї задачі з графами, що мають обмежену структуру). Крім того, для приведення використано екземпляр задачі 3-CNF-SAT, але не її рішення (не можна обґрунтовувати приведення за поліноміальний час на знанні того, чи виконувана формула φ, т.я. невідомо, як отримати цю інформацію за поліноміальний час).

Задача про вершинне покриття: знайти в заданому графі вершинне покриття мінімального розміру. У вигляді задачі прийняття рішення має вигляд: чи містить граф вершинне покриття заданого розміру *k*. Вершинним покриттям неорієнтованого графу G = (V, E) є така підмножина $V' \subseteq V$, що якщо $(u, v) \in$ E, то або $u \in V'$, або $v \in V'$, або справедливі обидва твердження (кожна вершина «покриває» інцидентні ребра, вершинне покриття графу G – множина вершин, що покривають всі ребра з множини E).

Розміром вершинного покриття називають кількість вершин, що в ньому міститься. Визначимо мову: VERTEX-COVER = $\{\langle G, k\rangle$: G – граф, що має вершинне покриття розміром $k\}$.

Теорема. Задача про вершинне покриття є NP-повною [1].

Доведення. Покажемо, що VERTEX-COVER ∈ NP. Для цього для заданих графу G = (V, E) та цілого числа k в якості сертифікату виберемо саме вершинне покриття $V' \subseteq V$. В алгоритмі верифікації перевіряється, чи $|V'| = k$, далі перевіряється для кожного ребра $(u, v) \in E$, чи $u \in V'$ або чи $v \in V'$. Вказане можна виконати за поліноміальний час.

Покажемо, що CLIQUE $\leq_P$ VERTEX-COVER, щоб показати NP-складність задачі. Приведення основується на понятті «доповнення» для графу (доповнення неорієнтованого графу G = (V, E) – це $\overline{G} = (V, \overline{E})$, $\overline{E}$ $\{(u,v) : u, v \in V, u \neq v,$ й $(u,v) \notin E\}$). Алгоритм приведення в якості вхідних даних отримує екземпляр $\langle G, k \rangle$ задачі про кліку. В алгоритмі приведення обчислюється доповнення $\overline{G} = (V, \overline{E})$ (можна виконати за поліноміальний час). Виходом алгоритму є екземпляр $\langle G, |V| - k \rangle$ задачі про вершинне покриття. Покажемо, що таке перетворення дійсно є приведенням: граф G містить кліку розміром k тоді і тільки тоді, коли граф $\overline{G} = (V, \overline{E})$ має вершинне покриття розміром $|V| - k$. Припустимо, що граф G містить кліку $V' \subseteq V$ розміром $|V'| = k$. Стверджуємо, що $V - V'$ – вершинне покриття графу $\overline{G} = (V, \overline{E})$. Нехай (u, v) – довільне ребро з множини $\overline{E}$. Тоді $(u, v) \notin E$, а тому хоча б одна з вершин u, v не належить множині V' (т.я. кожна пара вершин з V' з'єднана ребром, що входить до множини E). Еквівалентно – хоча б одна з вершин u, v належить множині $V - V'$, а тому ребро (u, v) покривається цією множиною. Т.я. ребро (u, v) вибране з множини E довільно, то кожне ребро з цієї множини покривається вершиною з множини $V - V'$. Тому множина $V - V'$ розміром $|V| - k$ утворює вершинне покриття графу $\overline{G}$. І навпаки, нехай $\overline{G}$ має вершинне покриття $V' \subseteq V$, $|V'| = |V| - k$. Тоді для всіх пар вершин $u, v \in V$ з $(u, v) \in \overline{E}$ маємо, що або $u \in V'$, або $v \in V'$, або справедливі обидва твердження. Звідси буде справедливим: для всіх пар вершин $u, v \in V$, якщо $u \notin V'$ та $v \notin V'$, то $(u, v) \in E$. Тобто $V - V'$ є клікою розміром $|V'| - |V| = k$. А тому задача про вершинне покриття є NP-складною.

Задача про гамільтонові цикли.

Теорема. Задача про гамільтонові цикли є NP-повною [1].

Доведення. Покажемо, що задача HAM-CYCLE належить класу NP. Для заданого графу G = (V, E) сертифікат задачі має вигляд послідовності, що складається з $|V|$ вершин, що утворюють гамільтонів цикл. В алгоритмі верифікації перевіряється, що до цієї послідовності кожна вершина входить лише 1 раз та що, якщо повторити першу вершину після останньої, утвориться цикл у графі G. Тобто, перевіряється, що кожна пара послідовних вершин з'єднана ребром й ребром з'єднані перша та остання вершини послідовності. Таку перевірку можна виконати за поліноміальний час.

Покажемо, що VERTEX-COVER $\leq_P$ HAM-CYCLE, щоб показати NP-складність задачі. Побудуємо для заданого неорієнтованого графу G = (V, E) та цілого числа k неорієнтований граф $G' = (V', E')$, в якому гамільтоновий цикл міститься тоді і тільки тоді, коли розмір вершинного покриття графу дорівнює k.

Використаємо структурний елемент, що є фрагментом графа, який забезпечує його визначені властивості, а саме, для кожного ребра $(u,v) \in$ E граф $G' = (V', E')$, що будується, містить одну копію такого структурного елементу W_{uv} (рис. 17.1). Позначимо кожну вершину структурного елементу W_{uv} як $[u,v,i]$ або $[v,u,i]$, $1 \leq i \leq 6$ (тому кожен структурний елемент містить 12 вершин та 14 ребер). Потрібні властивості графу забезпечуються внутрішньою структурою елементу та накладеним обмеженням на зв'язки між структурним елементом й іншою частиною графу $G' = (V', E')$. Так, назовні елементу W_{uv} виходитимуть ребра тільки з вершин $[u,v,1]$, $[u,v,6]$, $[v,u,1]$, $[v,u,6]$. Будь-який гамільтоновий цикл G' має проходити по ребрах структурного елементу W_{uv} одним з 3 способів (рис. 17.2). Якщо цикл проходить через вершину $[u,v,1]$, то вийти він має через вершину $[u,v,6]$, при цьому він або проходить через всі 12 вершин елементу, або тільки через 6 – з $[u,v,1]$ до $[u,v,6]$. Якщо тільки через 6 вершин, то цикл повинен вдруге зайти до структурного елементу й пройти через 6 вершин від $[v,u,1]$ до $[v,u,6]$ (рис. 17.2, а). Якщо цикл входить до структурного елементу че

рез вершину $[v,u,1]$, то аналогічно – він має пройти або через всі 12

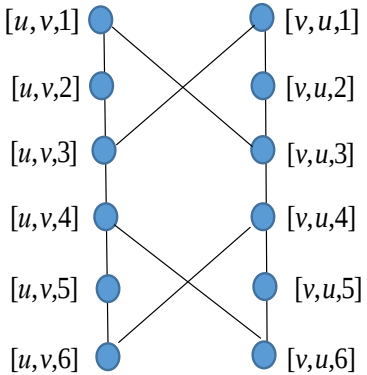


Рис. 17.1. Структурний елемент W_{uv}

вершин, або через 6, від $[v,u,1]$ до $[v,u,6]$ (для цього випадку цикл також повинен вдруге зайти до структурного елементу й пройти через 6 вершин) (рис. 17.2, б – в). Жодних інших варіантів проходження всіх 12 вершин структурного елементу немає. Тому неможливо побудувати 2 шляхи, що не перетинаються, один з яких з'єднує вершини $[u,v,1]$ та $[v,u,6]$, а другий – вершини $[v,u,1]$ та $[u,v,6]$, так, щоб об'єднання цих шляхів містило всі вершини структурного елементу.

Єдині вершини, що містяться у множині V', крім вершин структурних елементів, – це вершини, що переключають, s_1, s_2, …, s_k. Ребра графу G', інцидентні вершинам, що переключають, використовують для вибору k вершин з покриття графу G.

Разом з ребрами, що входять до складу структурних елементів, множина E' містить ребра двох інших типів. Перший. Для кожної вершини $u \in V$ додаються ребра, з'єднуючі пари структурних елементів у такому порядку, щоб одержати шлях, що містить усі структурні елементи, які відповідають ребрам, інцидентним вершині u графу G. Вершини, суміжні з кожною вершиною $u \in V$, впорядковуються довільно як $u^1, u^2, ..., u^{\deg ree(u)}$, де $\deg ree(u)$ - кількість вершин, суміжних з вершиною u. У графі G' створюється шлях, що проходить через всі структурні елементи, що відповідають інцидентним ребрам вершини u. Для цього до множини E' додаються ребра $\{([u,u^i,6], [u,u^{i+1},1])$:

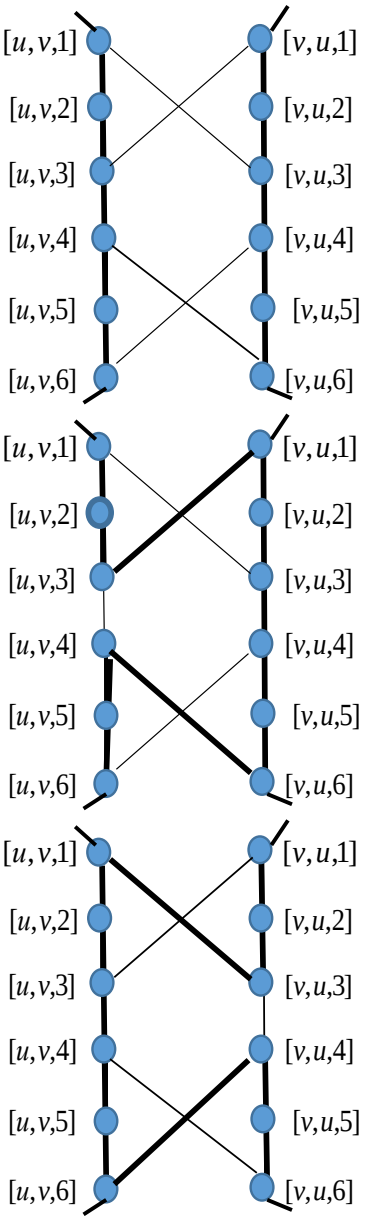


Рис. 17.2. Проходження гамільтонового циклу по ребрах W_{uv}

$1 \le i \le \deg ree(u) - 1\}$. Наприклад, вершини, суміжні з вершиною *w*, впорядковуються як *x, y, z*, так що граф G' містить ребра $([w, x, 6], [w, y, 1])$ та $([w, y, 6], [w, z, 1])$. Для кожної вершини $u \in V$ ці ребра графа G' заповнюють шлях, що містить всі структурні елементи, які відповідають ребрам, інцидентним вершині *u* графу G. Міркування на користь існування цих ребер наступні: якщо з вершинного покриття графу G вибирається вершина $u \in V$, то в графі G' можна побудувати шлях з вершини $[u, u^1, 1]$ до вершини $[u, u^{\deg ree(u)}, 6]$, що покриває всі структурні елементи, відповідні ребрам, інцидентним вершині *u* (тобто дя кожного з цих структурних елементів цей шлях проходить або по всіх 12 вершинах (якщо вершина *u* входить до вершинного покриття, а вершина u^i – ні), або тільки по 6 вершинам $[u, u^i, 1], \ldots, [u, u^i, 6]$ (якщо обидві вершини *u* та u^i входять до вершинного покриття). Другий тип. Ребра, що поєднують першу $[u, u^i, 1]$ та останню $[u, u^{\deg ree(u)}, 6]$ вершини кожного з цих шляхів з кожною з вершин, що переключають:

$$\{(s_j, [u, u^1, 1]) : u \in V \text{ й } 1 \le j \le k\} \cup \{(s_j, [u, u^{\deg ree(u)}, 6]) : u \in V \text{ й } 1 \le j \le k\}.$$

Покажемо, що розмір графу G' виражає поліноміальна функція від розміру графу G, а тому граф G' можна побудувати за поліноміальний від розміру графу G час. Множина вершин графу G' складається з вершин, що входять до складу структурних елементів, та вершин, що переключають. Кожен структурний елемент містить 12 вершин. Ще є $k \le |V|$ вершин, що переключає. Тому $|V'| = 12|E| + k \le 12|E| + |V|$ вершин. Множина ребер графа G' складається з ребер, що належать структурним елементам, ребер, що поєднують структурні елементи, та ребер, які поєднують вершини, що переключають, із структурними елементами. Кожен структурний елемент має 14 ребер, а всі - $14|E|$. Для кожної вершини $u \in V$ граф G' містить $\deg ree(u) - 1$ ребер між структурними елементами, сумарно по всім вершинам множини *V* маємо $\sum_{u \in V} (\deg ree(u) - 1) = 2|E| - |V|$ ребер, що поєднують структурні елементи. Ще по два ребра належить кожній парі, що складається з 1 вершини, що переключає, та 1 вершини множини *V*. Разом таких ребер $2k|V|$. Отже, загалом ребер у графі G':

$$|E'| = 14|E| + (2|E| - |V|) + 2k|V| = 16|E| + (2k-1)|V| \le 16|E| + (2|V|-1)|V|.$$

Щоб довести, що перетворення графу G у граф G' є приведенням, припустимо, що граф G містить вершинне покриття $V^* \subseteq V$ розміром k та $V^* = \{u_1,...,u_k\}$. Гамільтонів цикл графу G' утворюється включенням до нього ребер (для кожної вершини $u_1,...,u_k$). Включають ребра: $\{([u_j, u_j{}^i, 6], [u_j, u_j{}^{i+1}, 1]) : 1 \le i \le \deg ree(u_j) - 1\}$, які поєднують всі структурні елементи, що відповідають ребрам, інцидентним вершинам u_j; ребра, що містяться в цих структурних елементах, залежно від того, як покривається ребро (одною або двома вершинами множини V^*); ребра $\{(s_j, [u_j, u_j^1, 1]) : 1 \le j \le k\} \cup \{(s_{j+1}, [u_j, u_j^{\deg ree(u_j)}, 6]) : 1 \le j \le k-1\} \cup$ $\cup \{(s_1, [u_k, u_k^{\deg ree(u_k)}, 6])\}$.

Ці ребра утворюють цикл, що починається у вершині s_1, проходить через усі структурні елементи, що відповідають ребрам, інцидентним вершині u_1, йде до вершини u_2, далі через всі структурні елементи, що відповідають ребрам, інцидентним вершині u_2, і т.д., доки знову не повернеться до вершини s_1. Кожен структурний елемент проходять один раз або двічі (залежно від того, одна чи дві вершини множини V^* покривають відповідне йому ребро). Т.я. множини V^* є вершинним покриттям графу G,то кожне ребро з множини E інцидентне деякій вершині з V^*, і тому цикл проходить через усі вершини кожного структурного елементу графу G'. А т.я. він також проходить через всі вершини, що переключають, то цикл є гамільтоновим. І навпаки, нехай граф G' містить гамільтоновий цикл $C \subseteq E'$. Стверджуємо, що множина $V^* = \{u \in V : (s_i, [u, u^1, 1]) \in C$ для деякого $1 \le i \le k\}$ є вершинним покриттям графу G. Щоб довести це, розіб'ємо цикл C на максимальні шляхи, що починаються у деякій вершині s_i, що переключає, проходять через ребро $(s_i, [u, u^1, 1])$ для деякої вершини $u \in V$ та закінчуються у вершині, що переключає, s_j, не перетинаючи при цьому жодних інших вершин, що переключають. Такий шлях називають «покриваючим». За способом побудови графу G' кожен покриваючий шлях має починатися у деякій вершині s_j, включати ребро $(s_i, [u, u^1, 1])$ для деякої вершини $u \in V$, проходити через усі структурні елементи, що відповіда

ють ребрам з множини E, інцидентним вершині $u \in V$, закінчуватися у деякій вершині, що переключає, s_j. Позначимо такий покриваючий шлях p_u та включимо вершину u до множини V^*. Кожен структурний елемент, через який проходить шлях p_u, має бути структурним елементом W_{uv} або W_{vu} для деякої вершини $v \in V$. Якщо через структурний елемент проходить шлях p_u, то по його вершинах проходить 1 або 2 покриваючих шляхи. Якщо один, то ребро $(u, v) \in E$ покривається у графі G вершиною u. Якщо 2, то один з них – шлях p_u, а інший – p_v. Тому $v \in V^*$ та ребро $(u, v) \in E$ покривається вершинами u та v. Т.я. усі вершини з кожного структурного елемента проходяться деяким покриваючим щляхом, маємо, що кожне ребро з E покривається деякою вершиною з V^*. Отже, задача про гамільтонові цикли є NP-складною.

Задача про комівояжера. Моделюючи задачу у вигляді повного графу з n вершинами, комівояжеру потрібно зробити тур (який є гамільтоновим циклом), щоб побувати у кожному місті рівно 1 раз та завершити його у тому ж місті, з якого тур почався. З кожним переїздом з міста i до міста j пов'язана його вартість $c(i, j)$, що є цілим числом. Комівояжеру потрібно виконати тур так, щоб його загальна вартість була мінімальною.

Формальна мова для відповідної задачі прийняття рішення: TSP = $=\{\langle G, c, k\rangle$: G = (V, E) – повний граф, c – функція $V \times V \to \mathrm{N}, k \cup \mathrm{N}, G$ містить тур вартістю не більше $k\}$.

Теорема. Задача про комівояжера є NP-повною [1].

Доведення. Доведемо, що TSP належить класу NP. Як сертифікат для заданого екземпляру задачі використаємо послідовність n вершин, що складають тур. В алгоритмі верифікації перевіряється, що в цій послідовності всі вершини містяться рівно1 раз та сумуються вартості всіх ребер туру й перевіряється чи сума не перевищує k. Вказане можна виконати за поліноміальний час.

Покажемо, що HAM-CYCLE $\leq_P$ TSP, щоб показати NP-складність задачі. Нехай G = (V, E) – екземпляр задачі TSP, що будується наступним чином. Створимо повний граф $G' = (V, E')$, $E' = \{(i, j) : i, j \in V, i \neq j\}$. Функцію вартості c визначимо:

$$c(i, j) = \begin{cases} 0, & (i, j) \in E, \\ 1, & (i, j) \notin E. \end{cases}$$

Т.я. граф G – неорієнтований та в ньому відсутні петлі, то для всіх вершин $u \in V$ справедливо $c(u, u) = 1$. Тому екземпляром TSP є набір $\langle G', c, 0 \rangle$, який легко скласти за поліноміальний час. Покажемо, що граф G містить гамільтонів цикл тоді і тільки тоді, коли граф $G' = (V, E')$ включає тур, вартість якого не перевищує 0. Припустимо, що граф G містить гамільтонів цикл *h*. Кожне ребро циклу *h* належить множині E, тому його вартість у графі G' є 0. І навпаки, припустимо, що граф G' містить тур *h'*, вартість якого не перевищує 0. Т.я. вартість ребер графу G' є 0 або 1, вартість туру тур *h'* є 0, то вартість кожного ребра туру має дорівнювати 0. Тому тур *h'* містить тільки ребра з множини E. Отже, тур *h'* є гамільтоновим циклом у графі G.

Задача про суму підмножини. Задане скінчена множина S додатних чисел та цільове значення $t > 0$. Чи існує підмножина $S' \subseteq S$, сума елементів якого дорівнює *t*. Формальна мова для відповідної задачі прийняття рішення: SUBSET-SUM ={ $\langle S, t \rangle$: є $S' \subseteq S$, таке, що $t = \sum_{s \in S'} s$ }.

Теорема. Задача про суму підмножин є NP-повною [1].

Доведення. Доведемо, що SUBSET-SUM належить класу NP. Як сертифікат для заданого екземпляру $\langle S, t \rangle$ задачі виберемо підмножину S'. Перевірку рівності $t = \sum_{s \in S'} s$ в алгоритмі верифікації можна виконати за поліноміальний час.

Покажемо, що 3-CNF-SAT $\leq_P$ SUBSET-SUM, щоб показати NP-складність задачі. Якщо задана 3-CNF-формула φ, що залежить від змінних $x_1, x_2, x_3, \ldots, x_n$ та містить підвирази, в кожен з яких входить по 3 літерали, то в алгоритмі приведення будується такий екземпляр $\langle S, t \rangle$ задачі, що формула φ виконувана тоді і тільки тоді, коли існує підмножина S, сума елементів якого дорівнює *t*. Припустимо, що у формулі φ в жодному підвиразі не міститься одночасно змінна та її заперечення, а також, що кожна змінна входить хоч би в один підвираз. В процесі приведення в множині S створюється по два числа для кожної змінної x_i та для кожного C_j. Ці числа будуть створюватися у десятковій системі зчислення (всі будуть містити по $n + k$ цифр, кожна з яких відповідає змінній або підвиразу у дужках). Основа системи зчислення 10 має властивість, необхідну для недопущення переносу значень з

молодших розрядів до старших).

Множина S та значення t будуються так: присвоюється кожному розряду числа мітка, що відповідає змінній або підвиразу у дужках. Цифри, що знаходяться у k молодших розрядах, помічають підвиразами, а цифри у n старших розрядах – змінними. Всі цифри цільового значення t, що помічені змінними, дорівнюють 1, а помічені підвиразами – 4. Для кожної змінної x_i в множині S містяться два цілих числа – v_i, v'_i. В кожному з них у розряді з міткою x_i знаходиться значення 1, а в розрядах, що відповідають всім іншим змінним, – значення 0. Якщо літерал x_i входить у підвираз C_j, то цифра з міткою C_j у числі v_i дорівнює 1. Якщо ж підвираз C_j містить літерал $\neg x_i$, цифра з міткою C_j у числі v'_i дорівнює 1. Всі інші цифри, мітки яких відповідають іншим підвиразам, у числах v_i, v'_i дорівнюють 0.

Значення v_i, v'_i в множині S не повторюються (якщо $l \neq i$, то v_l, v'_l не можуть бути рівними v_i, v'_i в старших розрядах; за вказаними припущеннями жодні v_i, v'_i не можуть бути рівними у всіх k молодших розрядах; якби всі числа v_i, v'_i були рівними, то літерали x_i, $\neg x_i$ мали б входити до одного й того ж набору підвиразів у дужках, але за припущенням жоден з підвиразів не містить одночасно x_i та $\neg x_i$, й хоча б один з них входить до одного з підвиразів, а тому має існувати якийсь з підвиразів C_j, для якого v_i, v'_i різні).

Для кожного підвиразу C_j у множині S містяться два цілих числа s_j, s'_j. Кожне з них містить нулі в усіх розрядах, мітки яких відрізняються від C_j. В числі s_j цифра з міткою C_j дорівнює 1, в числі s'_j – ця цифра дорівнює 2. Такі цілі числа служать «фіктивними змінними», які використовуються, щоб отримати в кожному розряді, поміченому підвиразом, цільове значення 4.

Максимальна сума цифр в кожному з розрядів дорівнює 6, що й є у цифрах, мітки яких відповідають підвиразам (три одиниці від значень v_i, v'_i та 1 плюс 2 від значень s_j, s'_j). Тому значення інтерпретуються як числа у десятковій системі зчислення (щоб не було переносу із молодших розрядів до старших).

Таке приведення можна виконати за поліноміальний час. Множина S містить $2n + 2k$ значень, в кожному з яких по $n + k$ цифр, тому час, необхідний для отримання всіх цифр, виражається поліноміальною функцією від $n + k$. Цільове значення t містить $n + k$ цифр, а в процесі приведення кожну з них можна отримати за фіксований час.

Покажемо, що 3-CNF-формула φ виконувана тоді і тільки тоді, коли існує підмножина $S' \subseteq S$, сума елементів якої дорівнює t. Нехай у формули φ є виконуючий набір. Якщо в ньому $x_i = 1$ ($i = 1, 2, \ldots, n$), то число v_i включається до множини S', інакше до S' включається число v'_i (до множини S' включаються саме ті значення v_i, v'_i, яким у виконуючому наборі відповідають літерали, що дорівнюють 1). Включаючи до S' v_i або v'_i та розміщуючи у значеннях s_j, s'_j нулі в усі розряди з мітками, що відповідають змінним, маємо, що сума цифр в цих розрядах по всім значенням множини S' має бути рівною 1, що співпадає з цифрами в цих розрядах цільового значення t. Т.я. всі підвирази у дужках виконуються, в кожному такому підвиразі є літерал, значення якого дорівнює 1. Тому кожна цифра, помічена підвиразом, включає хоча б одну 1, що вноситься до суми завдяки вкладу значення v_i або v'_i з множини S'. В дійсності у кожному підвиразі 1 можуть бути рівними 1,2, або 3 літерали, тому у кожному розряді, що відповідає підвиразу, сума цифр за значеннями v_i та v'_i множини S' дорівнює 1,2, або 3. Т.я. сума цифр за всіма значеннями множини S' співпадає з відповідними цифрами цільового значення t, а при сумуванні не було переносу значень з молодших розрядів до старших, то сума значень множини S' дорівнює t.

Припустимо, що є $S' \subseteq S$, сума елементів якої дорівнює t. Для кожного значення $i = 1, 2, \ldots, n$ ця підмножина повинна включати по одному із значень v_i, v'_i (інакше сума цифр в розрядах, що відповідають змінним, не була б рівною 1). Якщо $v_i \in S'$, то виконується присвоєння $x_i = 1$, інакше $v'_i \in S'$, виконується присвоєння $x_i = 0$. Доведемо, що за результатом такого присвоєння виконується кожен підвираз C_j, $j = 1, 2, \ldots, k$. Маємо, щоб сума цифр у розряді з міткою C_j була рівною 4, підмножина S' має включати хоча б одне із значень v_i, v'_i, в якому 1 знаходиться у розряді з міткою C_j, так як сумарний вклад фіктивних змінних s_j, s'_j не перевищує 3. Якщо підмножина S' включає значення v_i, в якому у розряді з міткою C_j міститься 1, то у підвираз C_j входить літерал x_i. Т.я. при $v_i \in S'$ виконується присвоєння $x_i = 1$, підвираз C_j виконується. Якщо S' включає значення v'_i, в якому в цьому розряді міститься 1, то до підвиразу C_j входить літерал $\neg x_i$. Т.я. при $v'_i \in S'$ виконується присвоєння $x_i = 0$, знову підвираз C_j виконується. Тому у формулі φ виконуються всі підвирази.

Лекція 18. Наближені алгоритми. Оцінка якості наближених алгоритмів

Існує дуже багато NP-повних практичних задач і часто саме для таких задач важливо знайти можливість побудови за поліноміальний час рішення, близького до оптимального (у найгіршому або середньому випадку). Отже, розглянемо приклади побудови алгоритмів, що повертають рішення, близькі до оптимальних (такі алгоритми називають ***наближеними***).

Нехай маємо деяку задачу *оптимізації*, кожному з можливих рішень якої співставляється додатна вартість, й потрібно знайти рішення, близьке до оптимального. Залежно від задачі оптимальне рішення можна визначити як таке, якому відповідає максимально можлива (або мінімально можлива) вартість. Алгоритм рішення задачі має **відношення (коефіцієнт) апроксимації (наближення)** $\rho(n)$, якщо для довільних вхідних даних розміром n вартість C рішення, отриманого в результатів виконання цього алгоритму, відрізняється від вартості C^* оптимального рішення не більше ніж в $\rho(n)$ разів [1, 11]:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \le \rho(n).$$

Алгоритм, в якому досягається коефіцієнт апроксимації $\rho(n)$, називають $\rho(n)$-**наближеним алгоритмом**. Визначення коефіцієнта апроксимації та $\rho(n)$-наближеного алгоритму застосовувані до задач максимізації та мінімізації. Для задач максимізації виконується нерівність $0 < C \le C^*$, відношення C^*/C дорівнює величині, на яку вартість оптимального рішення більше вартості наближеного. Для задач мінімізації виконується нерівність $0 < C^* \le C$, відношення C/C^* показує у скільки разів вартість наближеного рішення більше вартості оптимального. Припускається, що вартості всіх рішень є додатними. Коефіцієнт апроксимації наближеного алгоритму не може бути меншим 1 (з нерівності $C/C^* < 1$ випливає $C^*/C > 1$). 1-наближений алгоритм видає опти мальне рішення, наближений алгоритм з більшим відношенням апроксимації може повернути рішення, що є набагато гіршим оптимального.

Схемою апроксимації називають наближений алгоритм, вхідні дані якого включають не тільки параметри екземпляру задачі, а й таке $\varepsilon > 0$, що для будь-якого фіксованого значення ε ця схема є $(1 + \varepsilon)$-набли

женим алгоритмом. Схему апроксимації називають **схемою апроксимації з поліноміальним часом виконання**, якщо для будь-якого фіксованого значення $\varepsilon > 0$ робота цієї схеми завершується за час, виражений поліноміальною функцією від розміру n вхідних даних. Час роботи схеми апроксимації з поліноміальним часом обчислення може швидко зростати за зменшення ε (наприклад, якщо час роботи схеми апроксимації оцінюється як $O(n^{2/\varepsilon})$.

Схема апроксимації є **схемою апроксимації з повністю поліноміальним часом роботи**, якщо час її роботи виражається поліномом як від $1/\varepsilon$, так і від розміру вхідних даних задачі n (наприклад, час роботи $O((1/\varepsilon)^2 n^3)$). В такій схемі будь-яке зменшення ε на сталий множник має відповідне зростання часу роботи на сталий множник.

Наведемо приклади побудови наближених алгоритмів з поліноміальним часом роботи та малими сталими відношеннями апроксимації, а для задачі про покриття множини – алгоритму з поліноміальним часом роботи, що характеризується коефіцієнтом апроксимації, величина якого зростає із зростанням розміру вхідних даних n.

Задача про вершинне покриття: знайти в заданому неорієнтованому графі вершинне покриття мінімального розміру (вершинне покриття графу G = (V, E) – множина вершин, що покривають всі ребра з множини E; розмір вершинного покриття – кількість вершин, що в ньому міститься). Вершинне покриття мінімального розміру називають *оптимальним вершинним покриттям*.

Наближений алгоритм APPOX-VERTEX-COVER [1] має вхідними даними параметри неорієнтованого графу G = (V, E), повертає вершинне покриття, розмір якого перевищує розмір оптимального вершинного покриття *не більше ніж у 2 рази*.

```
    APPOX-VERTEX-COVER(G)
1   C = ∅
2   E'= G.E
3   while E'≠∅
4       нехай (u, v) довільне ребро з E'
5       C = C ∪ {u, v}
6       вилучити з E' всі ребра, інцидентні u або v
7   return C
```

В процедурі APPOX-VERTEX-COVER змінна C містить створюване вершинне покриття (ініціалізується у рядку 1). У рядку 2 до множини E′ копіюється множина ребер G.E графу. У рядках 3-6 цикл по черзі вибирає ребра $(u, v) \in E'$, додає їх у кінцеві точки u, v в C та вилучає з E′ всі ребра, що покриваються або u, або v. У рядку 7 повертається вершинне покриття C.

Час роботи алгоритму APPOX-VERTEX-COVER – $O(V + E)$ при використанні для представлення E′ списків суміжності.

Теорема. APPOX-VERTEX-COVER є 2-наближеним алгоритмом з поліноміальним часом роботи [1].

Доведення. Час роботи алгоритму APPOX-VERTEX-COVER складає $O(V + E)$, тобто алгоритм має поліноміальний час роботи.

Множина вершин C, що повертається алгоритмом, є вершинним покриттям, так як алгоритм не виходить з циклу, доки кожне ребро G.E не буде покрите деякою вершиною з множини C. Покажемо, що алгоритм повертає вершинне покриття, розмір якого перевищує розмір вершинного покриття не більше ніж у 2 рази. Позначимо A множину ребер, що вибирається у рядку 4 процедури. Щоб покрити ребра множини A, кожне вершинне покриття має містити хоча б одну кінцеву точку кожного ребра x множини A. Жодні два ребра з цієї множини не мають спільних кінцевих точок, оскільки після вибору ребра у рядку 4 всі інші ребра з такими ж кінцевими точками вилучаються з множини E′ у рядку 6. Тоді жодні 2 ребра з множини A не покриваються одною й тою ж вершиною з множини C^*, а тому *нижня границя* розміру оптимального вершинного покриття дорівнює $|C^*| \geq |A|$. При кожному виконанні рядку 4 вибирається ребро, у якого жодна з кінцевих точок ще не увійшла до множини C (це дозволяє оцінити зверху розмір вершинного покриття, що повертається процедурою), тому: $|C| = 2|A|$. Звідси маємо: $|C| = 2|A| \leq 2|C^*|$ (тобто $|C| / |C^*| \leq 2$).

Задача про комівояжера. Маємо повний граф G = (V, E) з n вершинами. Знайти такий тур (який є гамільтоновим циклом), щоб побувати у кожній вершині саме 1 раз та завершити його у тому ж місті, з якого почався тур. З кожним ребром пов'язана його вартість $c(i, j)$, яка є цілим невід'ємним числом. Потрібно виконати тур так, щоб його загальна

вартість була мінімальною. Нехай c(A) – повна вартість всіх ребер підмножини $A \subseteq E$, тоді: c(A) = $\sum_{(u,v)\in A} c(u,v)$.

Найбільш дешевим переходом з одного місця до іншого є перехід по прямій. Якщо ж по дорозі зайти у деяке проміжне місце, то це не може привести до зменшення вартості (або якщо зрізати шлях й пропустити деякій проміжний пункт, це не приведе до зростання вартості). Отже, можемо ствердити, що функція вартості задовольняє нерівності трикутника, якщо для всіх вершин $u, v, w \in$ V:

$$c(u,w) \le c(u,v) + c(v,w).$$

Дійсно, наприклад, якщо вершини графа є точками на площині, а вартість переходу від одної вершини до іншої є євклідовою відстанню між точками, то нерівність трикутника виконується.

Обчислимо структуру – мінімальне остовне дерево – вага якої є нижньою границею довжини оптимального туру, щоб з її допомогою створити тур вартістю не більше ніж у 2 рази ваги остовного дерева. Виконує вказане процедура APPROX-TPS-TOUR(G, c) [1], в якій G – повний неорієнтований граф, c – функція вартості, що задовольняє нерівності трикутника.

```
    APPOX-TPS-TOUR(G, c)
1   вибрати довільно вершину r ∈ G.V в якості кореневої
2   викликати процедуру для обчислення мінімального
    остовного дерева T для графу G з кореня r
3   нехай H – список вершин, який впорядкований за
    моментом першого відвідування при обході дерева T
    у прямому порядку
4   return гамільтоновий цикл H
```

Виконуючи прямий обхід дерева рекурсивно відвідуємо всі його вершини (вершина заноситься до списку при першому її відвідуванні, до відвідування її дочкових вершин). Нехай маємо деякий повний неорієнтований граф. Обираємо деяку вершину в якості кореневої. Будуємо мінімальне остовне дерево T з коренем у цій вершині (використаємо процедуру MST-PRIM(G, c, r) [1]). Маємо, загальний час реалізації алгоритму – $\Theta(V^2)$.

Покажемо [1]: якщо функція вартості в екземплярі задачі про комівоя

жера задовольняє нерівності трикутника, то алгоритм APPROX-TPS-TOUR повертає тур, вартість якого не більше ніж у 2 рази перевищує вартість оптимального туру.

Спершу проілюструємо сказане. Нехай маємо повний неорієнтований граф вигляду як на рис. 18.1, (а) [1] (функція вартості в ньому – евклідова відстань між точками). Оберемо вершину *a* в якості кореневої (рис. 18.1, *б*) й побудуємо мінімальне остовне дерево з коренем у цій вершині. Виконаємо прямий обхід вздовж дерева (рис. 18.1, *в*), починаючи відвідування вершин з вершини *a*. В результаті отримаємо таку послідовність вершин: *a, b, c, b, h, b, a, d, e, f, e, g, e, d, a*.

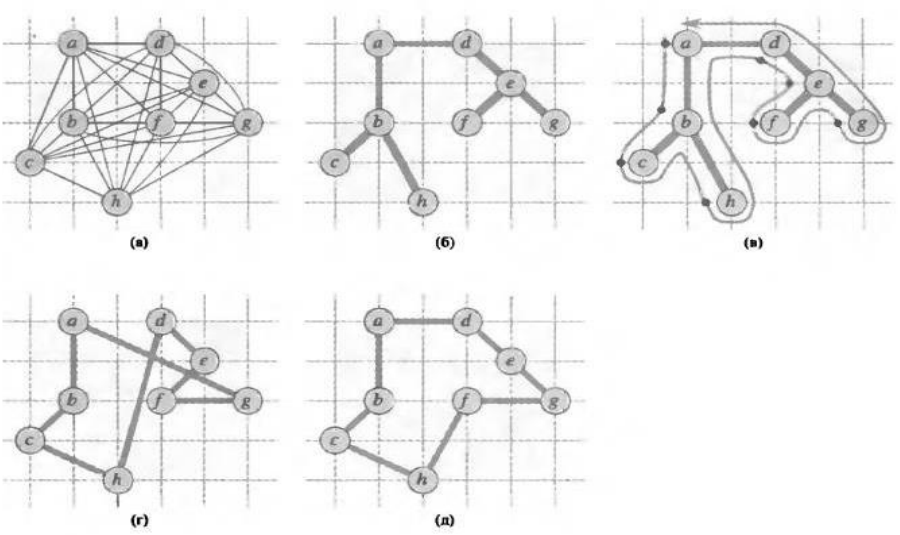


Рис. 18.1. Повний неорієнтований граф з мінімальним остовним деревом та турами [1].

Складемо список відвідування вершин, що відвідуються вперше. Він буде мати вигляд: *a, b, c, h, d, e, f,* g (рис. 18.1, *г*). В результаті матимемо, тур, що повертається алгоритмом APPROX-TPS-TOUR з вартістю приблизно 19,1 (ящо клітинку брати розміром 1×1). Оптимальний же тур для даного графу наведений на рис. 18.1 (*д*), його вартість – приблизно 14,7.

Теорема. Алгоритм APPROX-TPS-TOUR є 2-наближеним алгоритмом з поліноміальним часом роботи, що вирішує задачу про комівояжера за умови виконання нерівності трикутника [1].

Доведення. Час реалізації алгоритму – $\Theta(V^2)$, тобто APPROX-TPS-TOUR – алгоритм з поліноміальним часом роботи. Нехай H^* – тур, що є оптимальним для заданої множини вершин. Оскільки вилученням з цього туру одного ребра отримується остовне дерево, то вага мінімального остовного дерева T, що обчислюється у рядку 2 процедури APPROX-TPS-TOUR, дорівнює нижній границі вартості оптимального тура, тобто виконується $c(T) \le c(H^*)$.

При повному обході дерева T складається список вершин, які відвідуються вперше, а також коли до них відбувається повернення після відвідування піддерева. Позначимо такий обхід W. При повному обході получаємо відвідування вершин у деякому порядку. Оскільки при повному обході кожне ребро дерева T проходять рівно два рази, то справедливим буде: c(W) = 2c(T), а тому й $c(W) \le 2c(H^*)$. Отже вартість повного обходу W перевищує вартість оптимального туру не більше ніж у 2 рази. Але він не є туром, т.я. відбувається відвідування деяких вершин більше 1 разу. Проте за нерівністю трикутника відвідування будь-якої вершини можна відмінити й не збільшити вартість. Виконуючи таке неодноразово з обходу W можна виключити всі повторні відвідування для кожної вершини. В результаті отримаємо впорядкований список вершин, що співпадає з тим, який маємо при прямому обході дерева T. Нехай H – цикл, що відповідає даному прямому обходу. Це гамільтонів цикл, оскільки кожна вершина відвідується лише 1 раз. Саме цей цикл й обчислюється процедурою APPROX-TPS-TOUR. Оскільки цикл H отримується шляхом вилучення вершин з повного обходу W, то отримуємо: $c(H) \le c(W)$. А тому: $c(H) \le 2c(H^*)$.

Алгоритм APPROX-TPS-TOUR має коефіцієнт апроксимації 2, але алгоритм не є найкращим. Існує інший наближений алгоритм, що дає кращі практичні результати.

Якщо ж відмовитися від припущення про те, що функція вартості задовольняє нерівності трикутника, то не можна знайти тури з гарним наближенням за поліноміальний час.

Теорема. Якщо P ≠ NP, то для сталої $\rho \geq 1$ не існує наближеного алгоритму з поліноміальним часом роботи та коефіцієнтом апроксимації ρ, який дозволяє вирішити задачу про комівояжера у загальному випадку [1].

Доведення. «Від супротивного». Нехай для деякого $\rho \geq 1$ існує наближений алгоритм з поліноміальним часом роботи та коефіцієнтом апроксимації ρ. Припустимо, що ρ – ціле. Покажемо, що за допомогою алгоритму A можна розв'язувати екземпляри задачі про гамільтоновий цикл за поліноміальний час. Оскільки за доведеним задача про гамільтоновий цикл є NP-повною, то з її розв'язуваності за поліноміальний час випливає, що P = NP.

Нехай G = (V, E) є екземпляром задачі про гамільтоновий цикл. Спробуємо визначити за допомогою деякого наближеного алгоритму A, чи містить граф G гамільтоновий цикл. Перетворимо цей граф в екземпляр задачі про комівояжера. Нехай $G' = (V, E')$ повний граф на множині V, $E' = \{(i, j) : i, j \in V,\ i \neq j\}$ та призначимо кожному ребру з $G' = (V, E')$ вартість $c(i, j) = \begin{cases} 1, & (i, j) \in E, \\ 1 + \rho|V|, & (i, j) \notin E. \end{cases}$

Представлення графу G' та функції вартості *c* можна отримати з представлення графу G за час, що поліноміально залежить від $|V|$ та $|E|$.

Розглянемо задачу (G', c). Якщо початковий граф G містить гамільтоновий цикл H, то функція вартості *c* співставляє кожному ребру циклу H одиничну вартість, а тому екземпляр задачі (G', c) містить тур вартістю $|V|$. Якщо ж граф G не містить гамільтоновий цикл H, то у будь-якому турі по графу G' має бути використаним деяке ребро (що не міститься у множині E), яке не менше величини

$$(\rho|V| + 1) + (|V| - 1) = \rho|V| + |V| > \rho|V|.$$

Внаслідок великої вартості ребер, що відсутні у графі G, між вартістю туру, що є гамільтоновим циклом у графі G (яка дорівнює $|V|$) та вартістю будь-якого іншого туру (що не менше $\rho|V| + |V|$) існує інтервал величиною не менше $\rho|V|$. Тому вартість туру, що не є гамільтоновим циклом у графі G, не менше ніж у ρ+1 разів більше вартості туру, що є гамільтоновим циклом у графі G.

Припустимо, що застосовуємо до екземпляру задачі про комівояжера (G', c) алгоритм А. Т.я. алгоритм А гарантовано повертає тур, вартість якого не більше ніж у ρ разів перевищує вартість оптимального туру (якщо граф G містить гамільтоновий цикл), то алгоритм А повинен повернути такий тур. Якщо G не містить гамільтонових циклів, то алгоритм А повертає такий тур, вартість якого перевищує $\rho|V|$. Тому за допомогою алгоритму А задачу про гамільтоновий цикл можна вирішити за поліноміальний час.

Наведене доведення є прикладом методики доведення, що задачу неможливо дуже добре апроксимувати.
Припустимо, що заданій NP-складній задачі X за поліноміальний час можна співставити таку задачу мінімізації У, що «так»-екземпляри задачі X будуть відповідати екземплярам задачі У, вартість, яких не перевищує деякого фіксованого *k*, а «ні»-екземпляри задачі X будуть відповідати екземплярам задачі У, вартість яких перевищує ρ*k*. Маємо показати: якщо тільки не виконується рівність P = NP, то не існує ρ-наближеного алгоритму з поліноміальним часом роботи, який може вирішити задачу У.

Задача про покриття множини у вигляді задачі прийняття рішення є узагальненням NP-повної задачі про вершинне покриття (тому також є NP-складною). Але наближений алгоритм, розроблений для задачі про вершинне покриття, не підходить для вирішення цієї задачі. Дослідимо простий евристичний жадібний метод з коефіцієнтом апроксимації, який виражається логарифмічною функцією (в ньому із зростанням розміру задачі може зростати розмір наближеного рішення відносно розміру оптимального рішення). Із зростанням розміру задачі якість розв'язку погіршується, але все ж досить повільно.
Екземпляр (X, F) задачі про покриття множини складається з кінцевої множини X та такої родини F підмножин множини X, що кожен елемент множини X належить хоча б одній підмножині з родини F:

$$X = \bigcup_{S \in F} S .$$

Вхідними даними задачі про покриття множини є скінченна множина U і родина F її підмножин. *Покриттям* називають родину C ⊆ F найменшої потужності, об'єднанням яких є U. На вхід подається пара

(U, F) та ціле число *k*; основним питанням задачі є існування покривної множини з потужністю *k* (або менше). Кажуть, що підмножина $C \in F$ покриває елементи, що в ньому містяться.
Наприклад, для множини *X* з 12 точок та родини $F = \{S_1, S_2, S_3, S_4, S_5, S_6\}$, зображених них на рис. 18.2 [1], матимемо наступне. Мінімальним покриттям є $C = \{S_3, S_4, S_5\}$ розміру 3. Жадібний алгоритм дасть покриття розміру 4, а саме $\{S_1, S_4, S_5, S_6\}$ або $\{S_1, S_4, S_5, S_3\}$.

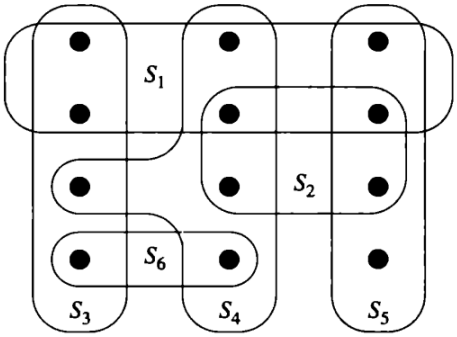


Рис. 18.2. Екземпляр (X, F) задачі про покриття заданої множини [1].

За умовами задачі маємо знайти підмножину $C \subseteq F$ мінімального розміру, члени якого покривають всю множину *X*, тобто $X = \bigcup_{S \in C} S$. Тоді кажуть, будь-яка родина C, що задовольняє вказаному виразу, покриває множину X. Розмір родини C визначається як кількість підмножин, що в ньому міститься.
Задача про покриття множини – жадібний наближений алгоритм.
Алгоритм GREEDY-SET-COVER [1] обирає на кожному кроці множину S, що покриває максимальну кількість елементів з тих, що залишилися непокритими.

```
    GREEDY-SET-COVER(X, F)
1   U = X
2   C = ∅
3   while U ≠ ∅
4       вибрати S ∈ F, що максимізує |S ∩ U|
```

```
5        U = U – S
6        C = C ∪{S}
7    return C
```

В алгоритмі на кожному кроці в множині U містяться елементи, що залишилися непокритими. Множина C містить покриття, побудоване алгоритмом. У рядку 4 маємо момент прийняття рішення жадібного підходу. Вибирається підмножина S, що покриває максимально можливу кількість ще покритих елементів (неоднозначності вирішуються довільно). Після вибору підмножини S її елементи вилучаються з множини U, а підмножину S розміщують у родину C. По закінченню роботи алгоритму множина C буде містити підродину родини F, що покриває множину X.

Наведену процедуру можна реалізувати так, щоб час її роботи виражався поліноміальною функцією від $|X|$ та $|F|$. Кількість ітерацій циклу (рядки 3-6) обмежено зверху величиною $\min(|X|,|F|)$. Вказаний цикл можна реалізувати за час $O(|X||F|)$, а увесь алгоритм за час, наприклад, $O(|X||F| \min(|X|,|F|))$.

Покажемо, що жадібний алгоритм повертає покриття множини, що не дуже перевищує оптимальне покриття.

Теорема. Алгоритм GREEDY-SET-COVER є $\rho(n)$-наближеним алгоритмом з поліноміальним часом роботи, $\rho(n) = H(\max\{|S| : S \in F\})$, де $H(d) = \sum_{i=1}^{d} 1/i$ [1].

Доведення. Вище показано, що алгоритм GREEDY-SET-COVER можна виконати за поліноміальний час.

Покажемо, що GREEDY-SET-COVER є $\rho(n)$-наближеним алгоритмом. Надамо кожній з обраних алгоритмом множин вартість 1, розподілимо її по всіх елементах, покритих за перший раз. За допомогою цих вартостей отримаємо шукане співвідношення між розміром оптимального покриття множини C^* та розміром повернутого алгоритмом покриття C. Позначимо i-ту підмножину, обрану алгоритмом, S_i. Додавання підмножини S_i до множини C збільшує вартість на 1. Рівномірно розподілимо вартість, що відповідає вибору підмножини S_i, між елементами,

що вперше покриваються цією підмножиною. Позначимо c_x вартість, що виділена елементу x, $x \in X$. Вартість виділяють кожному елементу лише один раз, коли цей елемент покривається вперше. Якщо елемент x перший раз покривається підмножиною S_i, то виконується

$$c_x = \frac{1}{\left|S_i - (S_1 \cup S_2 \cup ... \cup S_{i-1})\right|}.$$

На кожному кроці алгоритму присвоюємо одиничну вартість, тому $\left|C\right| = \sum_{x \in X} c_x$. Кожний елемент $x \in X$ знаходиться як мінімум в одній множині з оптимального покриття C^*, тому $\sum_{S \in C^*} \sum_{x \in S} c_x \geq \sum_{x \in X} c_x$. Звідси

$$\left|C\right| \leq \sum_{S \in C^*} \sum_{x \in S} c_x .$$

Далі, т.я. для довільної множини $S \in F$ виконується $\sum_{x \in S} c_x \leq H(\left|S\right|)$, то

$$\left|C\right| \leq \sum_{S \in C^*} H(\left|S\right|) \leq \left|C^*\right| H(\max\{\left|S\right| : S \in F\}).$$

Покажемо, що дійсно $\sum_{x \in S} c_x \leq H(\left|S\right|)$. Розглянемо довільну множину $S \in F$ та індекс $i = 1, 2, \ldots, \left|C\right|$ та уведемо $u_i = \left|S_i - (S_1 \cup S_2 \cup ... \cup S_{i-1})\right|$, що дорівнює кількості елементів множини S, які залишилися непокритими після того, як у алгоритмі були вибрані множини $S_1, S_2, \ldots, S_i$. Визначимо $u_0 = \left|S\right|$, рівну кількості елементів множини S, що початково є непокритими. Нехай k – мінімальний індекс, за якого виконується рівність $u_k = 0$, тобто кожен елемент S покривається хоча б однією з множин $S_1, S_2, \ldots, S_k$. Тоді $u_{i-1} \geq u_i$ й за $i = 1, 2, \ldots, k$ множиною S_i вперше покриваються $u_{i-1} - u_i$ елементів множини S. Тоді

$$\sum_{x \in S} c_x = \sum_{i=1}^{k} (u_{i-1} - u_i) \frac{1}{\left|S_i - (S_1 \cup S_2 \cup ... \cup S_{i-1})\right|}.$$

Маємо $\left|S_i - (S_1 \cup S_2 \cup ... \cup S_{i-1})\right| \geq \left|S - (S_1 \cup S_2 \cup ... \cup S_{i-1})\right| = u_{i-1}$, т.я. за жадібного вибору S_i гарантується, що множина S не може покрити більше нових елементів, ніж множина S_i (інакше замість множини S_i була б обраною множина S). Тому

$$\sum_{x \in S} c_x \leq \sum_{i=1}^{k} (u_{i-1} - u_i) \frac{1}{u_{i-1}}.$$

Далі (враховуючи, що $H(0) = 0$ та $j \le u_{i-1}$)

$$\sum_{x \in S} c_x \le \sum_{i=1}^{k} (u_{i-1} - u_i) \frac{1}{u_{i-1}} = \sum_{i=1}^{k} \sum_{j=u_{i+1}}^{u_{i-1}} \frac{1}{u_{i-1}} \le \sum_{i=1}^{k} \sum_{j=u_{i+1}}^{u_{i-1}} \frac{1}{j} =$$

$$= \sum_{i=1}^{k} \left(\sum_{j=1}^{u_{i-1}} \frac{1}{j} - \sum_{j=1}^{u_i} \frac{1}{j} \right) = \sum_{i=1}^{k} (H(u_{i-1}) - H(u_i)) =$$

$$= H(u_0) - H(u_k) = H(u_0) - H(0) = H(u_0) = H(|S|).$$

Наслідок. Алгоритм GREEDY-SET-COVER є $(\ln|X| + 1)$-наближеним алгоритмом з поліноміальним часом роботи [1].

Іноді величина $\max\{|S| : S \in F\}$ є невеликою сталою, тому рішення, що повертається алгоритмом, більше оптимального на множник, не перевищуючий малу сталу.

Задача про суму підмножин. Для заданої скінченої множини S додатних чисел, цільового цілого значення $t > 0$ визначити чи існує підмножина $S' \subseteq S$, сума елементів якої дорівнює t. Екземпляр задачі має вигляд пари $\langle S, t \rangle$.

Задача про суму підмножин є NP-повною.

1. Точний алгоритм з експоненціальним часом роботи. Нехай для кожної підмножини $S' \subseteq S$ обчислюють суму елементів, а потім з усіх підмножин, сума елементів яких не перевищує t, вибирають підмножину, для якої ця сума найменше відрізняється від t. Вказаний алгоритм повертає оптимальне рішення, але час його роботи буде показниковою функцією. Для його реалізації можна скористуватися ітеративною процедурою, в якій в i-й ітерації обчислюють суми елементів всіх підмножин множини $\{x_1, x_2, \ldots, x_i\}$ за раніше обчисленими сумами всіх підмножин множини $\{x_1, x_2, \ldots, x_{i-1}\}$. В такому алгоритмі не потрібно продовжувати виконувати обробку деякої підмножини S' після того, як сума її елементів стане більшою за t. Вхідними даними відповідної процедури EXACT-SUBSET-SUM є множина $S = \{x_1, x_2, \ldots, x_n\}$ та цільове значення t. В процедурі ітеративно обчислюється список L_i, що містить суми всіх підмножин множини $\{x_1, x_2, \ldots, x_n\}$, що не перевищують t. Потім повертається максимальне значення із списку L_n.

Якщо L – список додатних цілих чисел, а x – ще одне додатне ціле число, то L+ x – список цілих чисел, що отримані із списку L збільшенням кожного його елементу на x. Отже,

$$S + x = \{s + x : s \in S\}.$$

Використаємо допоміжну процедуру MERGE-LISTS(L, L′) [1] для сортування списку, який є об'єднанням вхідних відсортованих списків L, L′ з виключенням значень, які повторюються. Час виконання процедури MERGE-LISTS(L, L′) є $O(|L| + |L'|)$ (як і у процедури сортування злиттям).

```
    EXACT-SUBSET-SUM(S, t)
1   n = |S|
2   L0 = ⟨0⟩
3   for i = 1 to n
4       Li = MERGE-LISTS(Li-1, Li-1 +xi)
5       вилучити з Li всі елементи, що перевищують t
6   return найбільший елемент в Ln
```

Позначимо P_i множину всіх значень, які можна отримати вибравши підмножину множини $\{x_1, x_2, \ldots, x_i\}$ (можливо пусту) та склавши всі її елементи. Використавши, що $P_i = P_{i-1} \cup (P_{i-1} + x_i)$, методом математичної індукції по i можна показати, що L_i – відсортований список, який містить всі елементи множини P_i, що не перевищують t. Оскільки довжина списку L_i може досягати величини 2^i, то у загальному випадку час виконання алгоритму веде себе як показникова функція. А у випадках, коли t є поліномом від $|S|$ або всі числа, що містяться у множині S, обмежені зверху поліноміальними величинами від $|S|$, час виконання алгоритму буде поліноміально залежати від $|S|$.

2. Схема апроксимації з повністю поліноміальним часом роботи. Така схема дозволяє отримати наближене рішення задачі. Її можна скласти шляхом «скорочення» кожного списку L_i після його створення. Якщо деякі два значення у списку мало відрізняються один від одного, то для отримання наближеного рішення не доцільно підтримувати обидва ці

значення. Для цього використовується *параметр скорочення* δ, $0 < \delta < 1$. Щоб скоротити список L за параметром δ, необхідно вилучити із списку максимальну кількість елементів так, щоб в отриманому після скорочення списку L' для кожного вилученого із списку L елементу y містився елемент z, що апроксимує елемент y, тобто $\frac{y}{1+\delta} \le z \le y$.

Вважатимемо, що елемент z представляє елемент y в новому списку L', тобто кожен вилучений елемент y представлений елементом z, що задовольняє вказаній нерівності. Наприклад: $\delta = 0{,}1$, L = ⟨10, 11, 12, 15, 20, 21, 22, 23, 24, 29⟩. Тоді L'= ⟨10, 12, 15, 20, 23, 29⟩.

Процедура TRIM [1] за заданими L та δ скорочує список L за час $\Theta(m)$, де m – довжина списку L, що відсортований у неспадаючому порядку. На виході маємо скорочений відсортований список L'.

```
    TRIM(L, δ)
1   Нехай m – довжина списку L
2   L' = ⟨y1⟩
3   last = y1
4   for i = 2 to m
5       if yi > last(1+ δ) // yi ≥ last (список L відсортований)
6           додати yi у кінець списку L'
7           last = yi
8   return L'
```

Елементи списку L скануються у неспадному порядку. Елемент розміщується у список L', якщо це перший елемент списку L або якщо його не можна представити останнім розміщеним у списку L' елементом.

Використовуючи процедуру TRIM схему апроксимації можна побудувати так. Вхідними даними процедури є множина $S=\{x_1, x_2, \ldots, x_n\}$, що є множиною цілих додатних чисел, записаних у довільному порядку, цільове ціле значення t та параметр апроксимації ε, $0 < \varepsilon < 1$. Процедура повертає значення z, величина якого відрізняється від оптимального не більше ніж у $1 + \varepsilon$ разів.

APPROX-SUBSET-SUM(S, t, ε)

1 n = $|S|$

2 $L_0 = \langle 0 \rangle$

3 **for** i = 1 **to** n

4 L_i = MERGE-LISTS(L_{i-1}, $L_{i-1} + x_i$)

5 L_i = TRIM(L_i, $\varepsilon/2n$)

6 вилучити з L_i всі елементи, що перевищують t

7 нехай z^* – найбільше значення в L_n

8 **return** z^*

В процедурі APPROX-SUBSET-SUM у рядку 2 список L_0 ініціалізується так, щоб у ньому містився тільки елемент 0. У циклі **for** (рядки 3-6) обчислюється відсортований список L_i, що містить скорочену версію множини P_i, з якої вилучені всі елементи, що перевищують значення t. Т.я. список L_i створюється на основі списку L_{i-1}, то маємо гарантувати, що повторне скорочення не вносить великих неточностей. Нехай маємо екземпляр задачі $S = \langle 104, 102, 201, 101 \rangle$ з $t = 308$, $\varepsilon = 0{,}40$, параметром скорочення $\delta = \varepsilon/8 = 0{,}05$. Процедура APPROX-SUBSET-SUM обчислить:

Рядок 2: $L_0 = \langle 0 \rangle$,

Рядок 4: $L_1 = \langle 0, 104 \rangle$,

Рядок 5: $L_1 = \langle 0, 104 \rangle$,

Рядок 6: $L_1 = \langle 0, 104 \rangle$,

Рядок 4: $L_1 = \langle 0, 102, 104, 206 \rangle$,

Рядок 5: $L_1 = \langle 0, 102, 206 \rangle$,

Рядок 6: $L_1 = \langle 0, 102, 206 \rangle$,

Рядок 4: $L_1 = \langle 0, 102, 201, 206, 303, 407 \rangle$,

Рядок 5: $L_1 = \langle 0, 102, 201, 303, 407 \rangle$,

Рядок 6: $L_1 = \langle 0, 102, 201, 303 \rangle$,

Рядок 4: $L_1 = \langle 0, 101, 102, 201, 203, 302, 303, 404 \rangle$,

Рядок 5: $L_1 = \langle 0, 101\ 201, 302, 404 \rangle$,

Рядок 6: $L_1 = \langle 0, 101, 201, 302 \rangle$.

Алгоритм поверне значення z* = 302, що наближає оптимальне рішення 307 = 104 + 102 +101 в межах ε (тобто фактична похибка 2%).

Теорема. Процедура APPROX-SUBSET-SUM є схемою апроксимації з повністю поліноміальним часом роботи та дозволяє вирішити задачу про суму підмножини [1].

Доведення. Після скорочення списку L_i у рядку 5 та вилучення з цього списку всіх елементів, що перевищують *t*, підтримується властивість: кожен елемент списку L_i є також елементом списку P_i. Тому значення z*, що повертається у рядку 8, є сумою деякої підмножини множини S. Позначимо оптимальне рішення задачі про суму підмножини $y^* \in P_n$.

З рядку 6 відомо, що $z^* \le y^*$. За нерівністю $\frac{y}{1+\delta} \le z \le y$ маємо показати, що $z^* / y^* \le 1+\varepsilon$. Крім того, покажемо, що час роботи алгоритму виражається поліноміальною функцією від 1/ε та від розміру вхідних даних. Для цього оцінюється границя довжини списку L_i: після скорочення послідовні елементи z та z' у списку L_i повинні задовольняти $z'/z > 1+\varepsilon/2n$, тобто відрізнятися не менше ніж у $1+\varepsilon/2n$ разів – тому кожен список містить 0, можливо 1 та до $\lfloor \log_{1+\varepsilon/2n} t \rfloor$ додаткових значень, а кількість елементів у кожному списку L_i не перевищує

$$\log_{1+\varepsilon/2n} t + 2 = \frac{\ln t}{\ln(1+\varepsilon/2n)} + 2 \le \frac{2n(1+\varepsilon/2n)\ln t}{\varepsilon} + 2 < \frac{3n \ln t}{\varepsilon} + 2$$

(т.я. $0 < \varepsilon < 1$, $\frac{x}{1+x} \le \ln(1+x) \le x$). Отримана границя є поліноміальною функцією від розміру вхідних даних (сума кількості бітів $\log_2 t$, що необхідні для представлення числа *t*, та кількості бітів, необхідних для представлення множини S, яка поліноміально залежить від *n* та 1/ε). Отже, оскільки час виконання процедури виражається поліноміальною функцією від довжини списку L_i, то процедура є схемою апроксимації з поліноміальним часом роботи.

Лекція 19. Наближені алгоритми. Методи побудови наближених алгоритмів

Рандомізований алгоритм рішення задачі має **коефіцієнт апроксимації** $\rho(n)$, якщо для довільних вхідних даних розміру n очікувана вартість С розв'язку, отриманого з допомогою цього рандомізованого алгоритму, не більше ніж у $\rho(n)$ разів перевищує вартість C^* оптимального розв'язку:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \le \rho(n) \cdot$$

Рандомізований алгоритм, що дозволяє отримати коефіцієнт апроксимації $\rho(n)$, називають *рандомізованим $\rho(n)$-наближеним алгоритмом*.

Розглянемо побудову *рандомізованого наближеного* алгоритму на прикладі вирішення задачі про MAX-3-CNF-виконуваність. У задачі 3-CNF-виконуваності запитується, чи виконувана задана формула φ, що належить класу 3-CNF (задача, як відомо, є NP-повною). Якщо конкретний екземпляр задачі є виконуваним, то існує такий варіант присвоєння змінних, за якого кожний підвираз у дужках, що складається з трьох літералів, приймає значення 1. Якщо ж екземпляр задачі не є виконуваним, то може виникнути необхідність в оцінці, наскільки цей екземпляр є близьким до виконуваного (у пошуку, які значення потрібно надати змінним, щоб виконалася максимально можлива кількість підвиразів у дужках). Отриману таким чином задачу називають задачею про MAX-3-CNF-виконуваність, її вхідні дані співпадають із вхідними даними задачі про 3-CNF-виконуваність.

Отже, у задачі про MAX-3-CNF-виконуваність потрібно знайти такі надані значення змінних, за яких значення 1 приймається у максимальній кількості підвиразів у дужках. Покажемо, що якщо значення присвоювати кожній змінній випадковим чином (значення 0 та 1 надаються із ймовірністю ½), то отримаємо рандомізований 8/7-наближений алгоритм. Припустимо, що жоден з підвиразів у дужках не містить одночасно змінної та її заперечення.

Теорема. Для заданого екземпляра задачі про MAX-3-CNF-виконуваність з n змінними $x_1, x_2, \ldots, x_n$ та m підвиразами в дужках рандомі-

зований алгоритм, в якому кожній змінній незалежно випадковим чином з ймовірностями ½ присвоюється значення 0 або 1 є рандомізованим 8/7-наближеним алгоритмом [1].

Доведення. Припустимо, кожній змінній незалежно випадковим чином з ймовірностями ½ присвоюється значення 0 або 1. Визначимо індикаторну випадкову величину Y_i = I{підвираз i виконується}, $i = 1, 2, \ldots, m$. Тоді $Y_i = 1$ виконується, якщо хоча б одному з літералів, що містяться в i-му підвиразі у дужках, присвоєне значення 1. Оскільки жоден з літералів не входить до одного й того ж підвиразу у дужках більше 1 разу та припущено, що жоден підвираз у дужках не містить одночасно змінну та її заперечення, то присвоєння значень трьом змінним у кожних дужках виконується незалежно. Підвираз у дужках не виконуватиметься тільки, коли всім трьом літералам присвоюється Y значення 0. Тоді Pr{підвираз i не виконується} = $(1/2)^3 = 1/8$. Звідси: Pr{підвираз i виконується} = $1 - 1/8 = 7/8$. Тоді за основною властивістю індикаторної випадкової величини: $E[Y_i] = 7/8$.

Позначимо: Y – загальна кількість підвиразів, що виконується ($Y = Y_1 + Y_2 + \ldots + Ym$). Тоді

$$E[Y] = E[\sum_{i=1}^{m} Yi\ ] = \sum_{i=1}^{m} E[Yi] = \sum_{i=1}^{m} 7/8 = 7m/8.$$

Маємо, верхня границя кількості підвиразів у дужках, що виконуватиметься, дорівнює m, тому коефіцієнт апроксимації не перевищує значення $m/(7m/8) = 8/7$.

Отримаємо алгоритм для апроксимації зваженого вершинного покриття у задачі про вершинне покриття з мінімальною вагою за допомогою лінійного програмування [1]. В задачі про вершинне покриття з мінімальною вагою (в якій шукають вершинне покриття з мінімальною вагою) задається неорієнтований граф G = (*V*, E), а кожній вершині $v \in V$ призначають додатну вагу $\omega(v)$. Вагу довільного вершинного покриття $V' \subseteq V$ визначають наступним чином: $\omega(V') = \sum_{v \in V'} \omega(v)$. Застосування алгоритму для пошуку незваженого вершинного покриття до цієї задачі не підходить, оскільки може дати рішення далеке від оптимального. Використання лінійного програмування дозволяє обчислити нижню границю ваги, яку й використаємо для отримання вершинного покриття.

Припустимо, що кожній вершині $v \in V$ співставляється змінна $x(v)$ та виконується умова: $x(v)$ дорівнює 0 або 1 для кожної $v \in V$. Рівність $x(v) = 1$ інтерпретуємо як належність вершини v до вершинного покриття, а рівність $x(v) = 0$ – як її відсутність у вершинному покритті. Обмеження, за якого для довільного ребра (u, v) хоча б одна з вершин (u або v) повинна входити до вершинного покриття можна задати нерівністю $x(u) + x(v) \geq 1$. Це приводить до отримання 0-1-цілочисельної задачі лінійного програмування пошуку вершинного покриття з мінімальною вагою [1]:

$$\sum_{v \in V} \omega(v) x(v) \to \min$$
$$x(u) + x(v) \geq 1 \text{ для кожного } (u, v) \in \mathrm{E},$$
$$x(v) \in \{0,1\} \quad \text{для кожної } v \in V.$$

У випадку, коли всі ваги $\omega(v) = 1$, отримується NP-складна оптимізуюча версія задачі про вершинне покриття.

Припустимо, що обмеження $x(v) \in \{0,1\}$ усувається, замість нього накладено $0 \leq x(v) \leq 1$. В результаті отримується послаблена задача лінійного програмування [1]:

$$\sum_{v \in V} \omega(v) x(v) \to \min$$
$$x(u) + x(v) \geq 1 \text{ для кожного } (u, v) \in \mathrm{E},$$
$$x(v) \leq 1 \quad \text{для кожної } v \in V,$$
$$x(v) \geq 0 \quad \text{для кожної } v \in V.$$

Крім того, будь-яке припустиме рішення отриманої 0-1-цілочисельної задачі лінійного програмування є також припустимим рішенням ослабленої задачі лінійного програмування. Тому оптимальне рішення задачі лінійного програмування є нижньою границею оптимального рішення 0-1-цілочисельної задачі. А тому й нижньою границею оптимального рішення задачі про вершинне покриття з мінімальною вагою.

Наведемо процедуру APPROX-MIN-WEIGHT-VC [1], що за допомогою розв'язання вказаної задачі лінійного програмування будує наближений розв'язок задачі про вершинне покриття з мінімальною вагою.

APPROX-MIN-WEIGHT-VC(G, ω)

1 C = $\emptyset$

2 Обчислення x, оптимального розв'язку задачі лінійного програмування

3 **for** кожної $v \in V$

4 **if** $x(v) \geq 1/2$

5 C = C $\cup \{v\}$

6 **return** C

У процедурі APPROX-MIN-WEIGHT-VC у рядку 1 ініціалізується вершинне покриття пустою множиною. У рядку 2 формулюється та вирішується вказана послаблена задача лінійного програмування. В оптимальному рішенні з кожною вершиною $v \in V$ зв'язане значення $0 \leq x(v) \leq 1$. У рядках 3-5, використовуючи вказане, визначається, які вершини додаються до вершинного покриття C. Якщо $x(v) \geq 1/2$, то вершина $v \in V$ додається до вершинного покриття C. Далі кожна дробова величина, що входить у рішення задачі лінійного програмування "заокруглюється" й отримується рішення 0-1-цілочисельної задачі лінійного програмування. У рядку 6 повертається вершинне покриття C.

Теорема. Алгоритм APPROX-MIN-WEIGHT-VC є 2-наближеним алгоритмом з поліноміальним часом роботи, який дозволяє вирішити задачу про вершинне покриття з мінімальною вагою [1].

Доведення. Існує алгоритм з поліноміальним часом роботи, що дозволяє вирішити задачу лінійного програмування (рядок 2) та цикл **for** (рядки 3-5) виконується за поліноміальний час, тому алгоритм APPROX-MIN-WEIGHT-VC є алгоритмом з поліноміальним часом роботи.

Покажемо, що він є 2-наближеним алгоритмом. Нехай: C^* – оптимальне рішення задачі про вершинне покриття з мінімальною вагою, z^* – значення оптимального рішення послабленої задачі лінійного програмування. Оскільки оптимальне вершинне покриття є припустимим рішенням цієї задачі, то z^* має бути нижньою гранню величини $\omega(C^*)$, тобто справедливе $z^* \leq \omega(C^*)$. В алгоритмі стверджується, що заокруглюючи дробові значення змінних $\bar{x}(v)$ отримується множина C, яка є вершинним покриттям, що задовольняє нерівності $\omega(C) \leq 2z^*$. Покажемо, що це дійсно так. Розглянемо довільне ребро $(u, v) \in E$. За обме

женням 0-1-цілочисельної задачі лінійного програмування має виконуватися: $x(u)+x(v)\geq 1$ для кожного $(u, v)\in$ E. Звідси випливає, що хоча б одна з величин $\bar{x}(u)$, $\bar{x}(v)$ не менше ½. Тому хоча б одна з вершин буде включена до вершинного покриття, а тому, будуть покриті всі ребра. Оцінимо вагу отриманого покриття:

$$z^* = \sum_{v\in V}\omega(v)\bar{x}(v) \geq \sum_{v\in V,\bar{x}(v)\geq 1/2}\omega(v)\bar{x}(v) \geq \sum_{v\in V,\bar{x}(v)\geq 1/2}\omega(v)\frac{1}{2} = \sum_{v\in C}\omega(v)\frac{1}{2} = \frac{1}{2}\sum_{v\in C}\omega(v) = \frac{1}{2}\omega(C).$$

Тоді ω(C) ≤ $2z^*$ ≤ $2\omega(C^*)$. Звідки випливає, що алгоритм APPROX-MIN-WEIGHT-VC є 2-наближеним алгоритмом.

Розглянемо побудову наближеного алгоритму для вирішення *задачі декількох комівояжерів* (задача про *m*-бродячих торговців, *m*-PSP). На відміну від класичної задачі про комівояжера (задачі максимізації гамільтонова циклу у неорієнтованому зваженому графі) в цій задачі два або більше комівояжерів. Розглядають також повний *n*-вершинний граф G = (*V*, E), а кожному ребру $e\in E$ призначають додатну вагову функцію $\omega: E\to R_+$, причому вага ребер належить [1, *q*]. Метою задачі є пошук *m* таких гамільтонових циклів H_1, …, H_m з множини E, що не перетинаються по ребрах й сумарна вага їх ребер є мінімальною або максимальною, тобто

$$\sum_{k=1}^{m}\sum_{e\in H_k}\omega(e)\to\begin{Bmatrix}\min\\ \max\end{Bmatrix}.$$

Досліджують різні варіанти вказаної задачі. Наприклад, для випадку двох комівояжерів, коли вага ребер належить інтервалу [1, *q*] та розглядають задачу максимізації, задача відома як 2-PSP-*max*. В ній шукають 2 таких гамільтонових цикла H_1, H_2, що не перетинаються ребрами, для яких сума $\omega_1(H_1) + \omega_2(H_2)$ є максимальною. Відомо, що навіть такий варіант задачі є NP-складною задачею [12]. Представимо ε-наближений алгоритм з поліноміальним часом роботи для задачі 2-PSP-*max* для $\omega_i\in\{1,2\}$. Для цього уведемо наступні поняття.

Неорієнтований граф G називають λ-*регулярним*, якщо степінь будь-якої його вершини дорівнює λ. *Цикловим покриттям графу* G називають його 2-регулярний підграф. Частковим туром у неорієнтованому графі G називають множину вершинно неперетинаючихся ланцюгів, що покривають всі вершини графу (ізольовану вершину вважають лан

цюгом довжини 0). Позначимо: OPT – граф $H_1^* \cup H_2^*$, ρ – доля ребер ваги 2 в OPT (число таких ребер ділене на $2n$), $\rho \in (0, 1)$, $\omega(\text{OPT}) = 2n(1+\rho)$.

Із задачею 2-PSP-*max*, $\omega_i \in \{1,2\}$ пов'язана задача 2-PSP-*min*, $\omega_i \in \{1,2\}$, для якої існують ε-наближені алгоритми з поліноміальним часом роботи (наприклад, ε = 4/3, 6/5, 11/7 та ін.). Крім того, показано [12], що для задачі 2-PSP-*max*, $\omega_i \in \{1,2\}$, існує наближений поліноміальний алгоритм рішення задачі з гарантованою оцінкою точності $1-\frac{7(\rho-1)}{18\rho-15}$.

Наприклад, побудова 6/5-наближеного алгоритму з поліноміальним часом для задачі 2-PSP-*min*, $\omega_i \in \{1,2\}$, використовує процедуру (позначають $P_{8/5}$), яка за $O(n^2)$ знаходить у n-вершинному 4-регулярному графі 2 часткових тури, що не перетинаються, із загальною кількістю ребер не менше $8n/5$. Далі ці тури добудовуються до двох гамільтонових циклів H_1, H_2, що не перетинаються по ребрах. Доведено, існує алгоритм A, який за $O(n^3)$ знаходить припустиме рішення задачі 2-PSP-*max*, $\omega_i \in \{1,2\}$, вага якого не менше 5/6 від ваги оптимальної пари циклів [12].

Розглянемо *процедуру доповнення часткового туру до гамільтонова циклу* ($P_{T\to H}$) для випадку гамільтонова циклу та туру, т.я. випадок двох турів можна звести до вказаного випадку. Нехай маємо гамільтоновий цикл H_1 та частковий тур T, що не перетинаються ребрами, у графі G. Вважаємо, що в T є не менше трьох ланцюгів додатної довжини (тур може містити й ланцюги довжини 0). Тоді процедура узагальнено може бути представлена так:

1. Доки у турі T є ізольована вершина, у T вибирати три довільні ланцюги додатної довжини (u_1, …, u_k), (z_1, …, z_l), (r_1, …, r_m). До T додати одне з ребер cu_1, cz_1, cr_1, яке не належить гамільтонову циклу H_1. Якщо вже є ланцюг довжини 0, то етап завершено.
2. Доки у турі T є більше 3 ланцюгів, вибирати 2 довільних ланцюги (u_1, …, u_k) та (z_1, …, z_l). До T додаємо одне з ребер u_1z_1, u_1z_l, u_kz_1, u_kz_l, що не належить циклу H_1.
3. На початок етапу тур T складається з трьох ланцюгів додатної довжини (u_1, …, u_k), (z_1, …, z_l), (r_1, …, r_m). З усіх можливих варіантів поєднання цих ланцюгів у гамільтоновий цикл H_2 виберемо той, що не використовує ребер циклу H_1.

Тоді за заданим гамільтоновим циклом H_1 та туром T, що не перетинаються ребрами, у графі G наведена процедура за час $O(n)$ будує гамільтоновий цикл $H_2 \supseteq$ T такий, що $H_1 \cap H_2 = \emptyset$.

Покажемо це. На етапі 1 є 3 ребра cu_1, cz_1, cr_1, тому хоч би одне з них не входить до H_1. На етапі 2 ребра u_1z_1, u_1z_l, u_kz_1, u_kz_l утворюють цикл довжини 4 (рис. 19.1). Тому хоча б одне з них не входить до H_1. Можливість побудови циклу H_2 на етапі 3 отримується розглядом можливих випадків розташування кінців ланцюгів, що залишаються на етапі 3, на циклі H_1. Отже, дійсно процедура за час $O(n)$ будує гамільтоновий цикл $H_2 \supseteq$ T такий, що $H_1 \cap H_2 = \emptyset$.

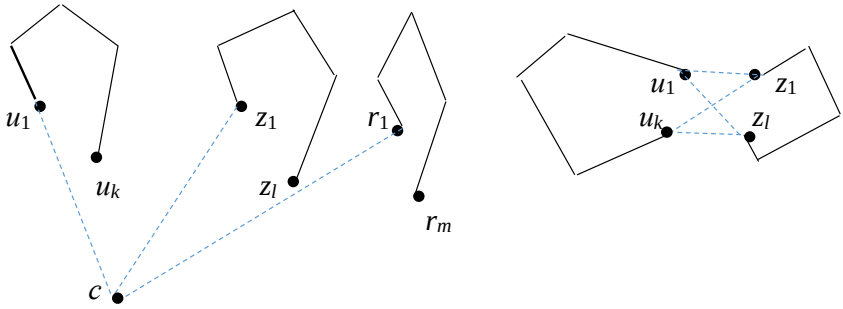


Рис. 19.1 Ребра етапів 1 та 2 процедури

Тепер розглянемо *алгоритм побудови часткового туру*. Нехай заданий n-вершинний граф G із степенями вершин 2, 3, 4. Процедуру, яку позначають $P_{2/3}$, у загальному вигляді можна представити так.

1. Доки є ребро $e = uz$, $e \in E$, в якому степені вершин u та z не менше 3, ребро вилучати з множини E. Після завершення етапу вершини із степенями 3, 4 стають несуміжними в G, їх відносять до множини V_1.

2. У графі G кожен ланцюг виду (u, z_1, ..., z_k, r) (цикл при $u = r$), де $k \geq 2, \deg(u) \geq 3, \deg(z_1) = \ldots = \deg(z_k) = 2, \deg(r) \geq 3,$ замінити на ланцюг (u, z, r), де z – нова вершина степені 2, а всі компоненти зв'язності графу G, що є простими циклами, вилучаються. При цьому можуть утворюватися мультиребра (при заміні циклу, тобто при $u = r$). При завершенні етапу вершини степені 2 стають несуміжними в графі

G, їх відносять до множини V_2. Отриманий граф $G' = (V_1, V_2, E')$ є дводольним мультиграфом.

3. Для кожної вершини з долі V_1 вибрати по 2 інцидентних для неї ребра. Отримується деякий набір T циклів та ланцюгів з кінцями у долі V_2.

4. Доки в T є цикли, вибирати довільну вершину $z \in V_2$ з деякого циклу та суміжну з нею по циклу вершину $u \in V_1$. Знаходиться таке ребро *ur*, яке не входить до туру T, що $\deg_T(r) \le 1$. Ребро *uz* в турі на ребро T замінити *ur*. Після завершення етапу T є частковим туром у G'.

5. У кожному простому циклі графу G, що був вилучений на етапі 2, вибрати простий ланцюг, що покриває всі його вершини, та додати до часткового туру T.

6. Для кожної вершини $z \in V_2$, що була отримана на етапі 2 з ланцюга (u, z_1, ..., z_k, r), до туру T додати ребра z_i, z_{i+1}, $1 \le i \le k-1$. Після завершення етапу тур T є частковим туром у початковому графі G.

Лема. У довільному графі G із степенями вершин 2, 3, 4 процедура $P_{2/3}$ за час $O(n)$ будує частковий тур T з $|T| \ge 2n/3$ ребрами [12].

Доведення. Спершу покажемо існування ребра *ur*, яке брали на етапі 4. Оскільки довільна вершина $u \in T \cap V_1$ має в G' степінь не меншу 3, то у $G' \setminus T$ є деяке ребро *ur*. А так як $\deg_{G'}(r) = 2$, то $\deg_T(r) \le 1$.

Знайдемо нижню оцінку для числа ребер у частковому турі T.

Спочатку розглянемо тур, отриманий після етапу 4 у графі G'. З обмежень на степені вершин у графі G' випливає, що $2|V_2| = |E'| \le 4|V_1|$.

Тому $|V_1| \ge \dfrac{|V_1| + |V_2|}{3} = |V(G')|/3$.

На етапі 3: $|T| = 2|V_1| \ge 2|V(G')|/3$. А на 4 етапі кількість ребер у турі T не змінюється. На етапі 5 для кожного k-вершинного циклу до T додають $k - 1$ ребро, що навіть для найгіршого випадку ($k = 3$) зберігає оцінку. На етапі 6 кількість доданих до G' вершин стає рівною числу доданих до туру T ребер, а тому оцінка зберігається.

Оцінимо час виконання процедури $P_{2/3}$. На етапі 1 переглядають всі ребра у графі G, на це витрачається час $O(|E(G)|)$. На етапі 2 для пошуку ланцюгів та компонент витрачається теж $O(|E(G)|)$. На етапі 3 формується початковий тур за $|V_1| \le n$ кроків. На етапі у тур перебудовується не

більше $|V_1| \le n$ разів. На етапах 5 та 6 до туру T додають не більше n ребер. Кожен з етапів виконується лише 1 раз. Звідси маємо асимптотичну оцінку $O(|E(G)|)$. З врахуванням нерівності $|E(G)| \le 2|V(G)| = 2n$ випливає оцінка $O(n)$.

Розглянемо алгоритм $A_{26/21}$ пошуку двох гамільтонових циклів [12], який у загальному вигляді опишемо так:

1. Деякою процедурою, яку назвемо P1, знайти в графі G такий гамільтоновий цикл H_1, що $\omega(H_1) \le \frac{7}{8}\omega(\text{OPT})/2$ (така процедура існує [12]).

2. Деякою процедурою, яку назвемо P2, знайти в графі G остовний регулярний степені 4 підграф G_4 з найменшою вагою ребер (така процедура існує [12]). З підграфу G_4 вилучити ребра циклу H_1. Отриманий граф позначити G'.

3. Процедурою $P_{2/3}$ знайти в графі G' частковий тур T, що містить не менше $2n/3$ ребер. Процедурою $P_{T\to H}$ добудувати отриманий частковий тур T до гамільтонова циклу H_2, що не перетинається з H_1.

Теорема. Алгоритм $A_{26/21}$ пошуку двох гамільтонових циклів за час $O(n^{24})$ знаходить таке рішення $H_1 \cup H_2$, що:

1. Якщо $\omega(\text{OPT}) = 2n$, то $\omega(H_1 \cup H_2) \le \frac{26}{21}\omega(\text{OPT})$ [12].

2. Якщо $\omega(\text{OPT}) > 2n$, то $\omega(H_1 \cup H_2) \le (\frac{11}{7} - \frac{1}{3(1+\rho)})\omega(\text{OPT})$.

Доведення. 1. За умовами маємо $\omega(\text{OPT}) = 2n$, тоді у графі G_4 є тільки ребра вагою 1. Вершини графу G' можуть мати тільки степені 2, 3, 4. Тому процедура $P_{2/3}$ є коректною та знаходить частковий тур T, для якого $|T| \ge 2n/3$. Звідси $\omega(H_2) \le |T| + 2(n - |T|) \le 4n/3$, а тому $\omega(H_1 \cup H_2) \le 8n/7 + 4n/3 = \frac{26}{21}\omega(\text{OPT})$. Асимптотична оцінка складності алгоритму $A_{26/21}$ є сумарною оцінкою етапів алгоритму. За [12] найвагомішим етапом є етап1, в якому процедура P1 потребує $O(n^{24})$ часу.

2. Якщо ω(OPT) > 2n, то у графі G_4 є не більше $2\rho n$ ребер ваги 2. У найгіршому випадку всі ці ребра належать до часткового туру T. Тому $\omega(H_2)$ с та $\omega(H_1) \le \frac{7}{8}\omega(\text{OPT})/2$. Звідси маємо:

$$\omega(H_1 \cup H_2) \le 4n/3 + 2\rho n + \frac{4}{7}\omega(\text{ОПТ}) = 2n(1+\rho) - 2n/3 + \frac{4}{7}\omega(\text{ОПТ}) \le$$

$$\le \left(\frac{11}{7} - \frac{1}{3(1+\rho)}\right)\omega(\text{OPT}).$$

Отже, й загальна асимптотична оцінка складності для наведеного наближеного алгоритму рішення задачі про двох комівояжерів буде $O(n^{24})$.

Рекомендована література

[1]. Кормен Т. Алгоритмы. Построение и анализ. 3-е изд. / Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн. – М. : ИД "Вильямс", 2013. – 1328 с.

[2]. Кнут Д. Искусство программирования, том 3. Сортировка и поиск. 3-е изд. / Д. Кнут. – М.: Вильямс, 2006. – С. 822.

[3]. Панос Л. Алгоритмы для начинающих: Теория и практика для разработчика. / Л. Панос. – М.: ЭКСМО, 2018. – 608 с.

[4]. Блейхут Р. Теория и практика кодов, контролирующих ошибки / Р. Блейхут. – М. : Мир, 1986. – 576 с.

[5]. Ross N. W. A painless guide to CRC error detection algorithms. – 1993. – 90 с. Ел. ресурс. Режим доступу: https://ceng2.ktu.edu.tr/~cevhers/ders_materyal/bil311_bilgisayar_mimarisi/supplementary_docs/crc_algorithms.pdf.

[6]. Тропченко А.Ю. Методы сжатия изображений, аудиосигналов и видео: Учебное пособие / А.Ю. Тропченко, А.А. Тропченко. – СПб: СПбГУ ИТМО, 2009. – 108 с.

[7]. Ватолин Д. Методы сжатия данных. Устройство архиваторов, сжатие изображений и видео / Д. Ватолин, А. Ратушняк, М. Смирнов, В. Юкин. – М.: ДИАЛОГ-МИФИ, 2003. – 384 с.

[8]. Потапов В.Н. Оценки избыточности кодирования последовательностей алгоритмом Лемпела-Зива // Дискретный анализ и исследование операций. – 1999. – Серия 1. – Т. 6. – № 2. – С. 70-81.

[9]. Сэломон Д. Сжатие данных, изображений и звука / Д. Сэломон. – М.: Техносфера, 2014. – 368 с.

[10]. Ульянов М. В. Ресурсноэффективные компьютерные алгоритмы / М. В. Ульянов. – М.: Наука, 2007. – 376 с.

[11]. Ковалев М.Я. Теория алгоритмов. Часть 2. Приближенные алгоритмы / М.Я. Ковалев, В.М. Котов, В.В. Лепин. – Минск: БГУ, 2002. – 155 с.

[12]. Гимади Э.Х., Истомин А.М., Рыков И.А. О задаче нескольких коммивояжёров с ограничениями на пропускные способности рёбер графа // Дискретный анализ и исследование операций. – 2013. – Т. 20. – № 5. – С. 13-30.

ЗМІСТ

Навчальне видання

Вергунова І.М.

Основи комп'ютерних алгоритмів.

Лекції

Підписано до друку 21.08.2021.
Формат 60х84/16. Папір офсетний.
Друк цифровий.
Друк. арк. 14,25. Умов. друк. арк. 13,25.
Обл.-вид. арк. 9,76.
Наклад 100 прим. Зам. № 5376/2.
Віддруковано з оригіналів замовника.
ФОП Корзун Д.Ю.
Свідоцтво про державну реєстрацію фізичної особи-підприємця
серія В02 № 818191 від 31.07.2002 р.

Видавець та виготовлювач ТОВ «ТВОРИ».
Свідоцтво про внесення суб'єкта видавничої справи
до Державного реєстру видавців, виготовлювачів і розповсюджувачів
видавничої продукції серія ДК № 6188 від 18.05.2018 р.
21034, м. Вінниця, вул. Немирівське шосе, 62а.
Тел.: 0 (800) 33-00-90, (096) 97-30-934, (093) 89-13-852,
(098) 46-98-043.
e-mail: info@tvoru.com.ua
http://www.tvoru.com.ua