Тема 12. Динамічне програмування

12.1. Задача складання розкладу зважених інтервальних робіт

Тепер ми перейдемо до більш потужної техніки розробки алгоритмів — динамічного програмування. Щоб краще зрозуміти та описати цей підхід, спочатку буде розглянуто приклад роботи динамічного програмування. Втім, загальна стратегія цього підходу схожа на метод декомпозиції: розв'язок задачі знаходиться шляхом поступового розбиття початкової задачі на дрібніші складові і послідовного вибудовування розв'язків для більших підзадач на основі підзадач меншої розмірності.

В попередній темі ми розглянули задачу складання розкладу інтервальних робіт (*Interval Scheduling Problem*), коли для заданого набору робіт (із відомим часом початку та тривалістю) необхідно обрати максимальну підмножину неконфліктних робіт (тих, які не перекривають одна одну). Тепер розглянемо одне узагальнення даної задачі. Припустимо, що кожна робота i має додаткову характеристику — вагу $v_i>0$. Цю характеристику можна інтерпретувати як грошову цінність, яку ми отримаємо, якщо включимо дану роботу в фінальний розклад. Ціль задачі, яку можна назвати складання розкладу зважених інтервальних робіт (*Weighted Interval Scheduling Problem*), тепер полягає в пошуку неконфліктної множини робіт з найбільшою сумарною вагою.

Дана постановка задачі є узагальненням попередньої, коли кожне $v_i = 1$. Але розв'язок цієї задачі вимагає більш складного підходу, аніж пропонується жадібним алгоритмом. Наприклад, якщо для задачі v_1 вага є більшою, аніж сума ваг усіх інших робіт, тоді оптимальним розв'язком буде включення цієї задачі v_1 незалежно від конфігурації інших інтервалів. Тож, будь-який алгоритм для цієї задачі повинен бути дуже чутливим до значень v_i. Як виявляється, не існує адекватного жадібного алгоритму для цієї задачі, тому тут ми використаємо підхід динамічного програмування.

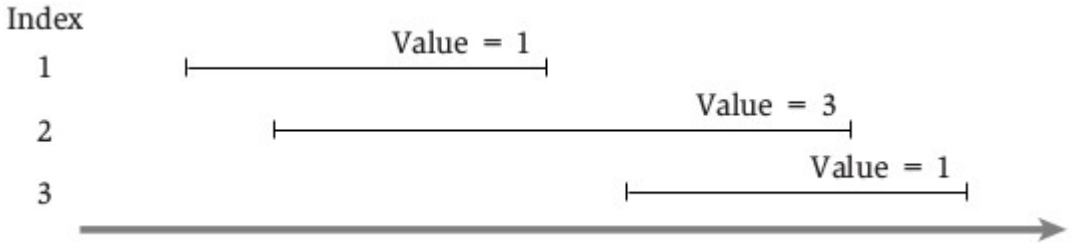


Рис. 12.1. Приклад задачі складання зваженого розкладу інтервальних робіт

Наведемо строге формулювання задачі. Задано n робіт, для кожної з яких визначено час початку s_i та час закінчення f_i. Кожний інтервал має вагу v_i. Два інтервали сумісні, якщо вони не перетинаються. Метою задачі є вибір підмножини $S \subseteq \{1, ..., n\}$ неконфліктних робіт із максимальним значенням суми ваг обраних інтервалів.

Припустимо, що роботи відсортовані у неспадному порядку їх закінчення: $f_1 \le ... \le f_n$. Будемо говорити, що робота i йде перед роботою j, якщо $i < j$. Це буде представляти природній порядок, в якому ми будемо надалі розглядати роботи. Також для роботи j позначимо через $p(j)$ найбільший індекс $i < j$, такий, що роботи i та j не перетинаються. Іншими словами, i — це крайня права робота, яка закінчується до початку роботи j. Визначимо $p(j) = 0$, якщо для j взагалі немає конфліктних робіт.

Тепер для довільного екземпляру задачі позначимо через S оптимальний розв'язок. Щодо S ми можемо стверджувати наступне: або остання робота n належить S, або ні. Якщо $n \in S$, то тоді жодна з робіт між $p(n)$ та n також не буде належати S, адже за означенням $p(n)$ всі роботи

$p(n) + 1$, $p(n) + 2$, ... , $n-1$ конфліктують з n. Більш того, якщо n ∈ S, тоді S має включати також оптимальний розв'язок для множини робіт {1, ... , $p(n)$}. З іншого боку, якщо $n \notin S$, то тоді S просто має включати оптимальний розв'язок для набору {1, ... , $n-1$}.

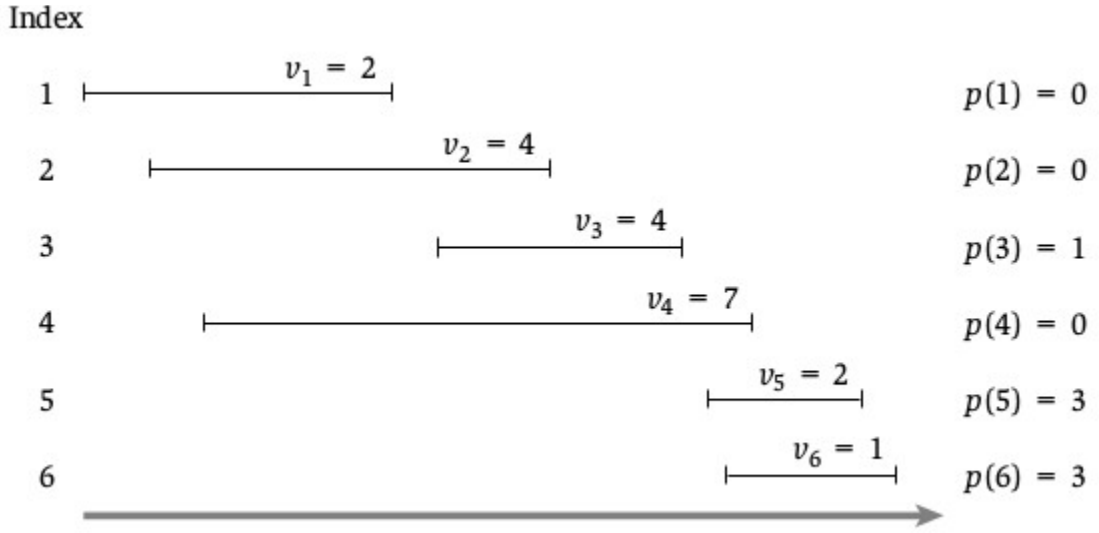


Рис. 12.2. Приклад зважених інтервальних робіт із значеннями $p(j)$ для кожної роботи.

Все це допускає, що пошук оптимального розв'язку для набору {1, ... , n} може бути представлений за допомогою пошуку оптимальних розв'язків задач меншої розмірності {1, 2, ... , j}. Тож, для будь-якого значення j між 1 та n позначимо через S_j оптимальне рішення для набору {1, 2, ... , j} і нехай $OPT(j)$ буде містити значення цього рішення. Введемо також граничне значення $OPT(0) = 0$ як оптимальне значення для порожнього набору робіт. Оптимальний розв'язок для всієї задачі тоді буде позначатись через S_n, а його значення — $OPT(n)$. Попередні наші міркування в нових термінах будуть означати наступне: якщо $j \in S_j$, то $OPT(j) = v_j + OPT(p(j))$; інакше якщо $j \notin S_j$, то $OPT(j) = OPT(j-1)$:

$$OPT(j) = \max\{v_j + OPT(p(j)), OPT(j-1)\} \quad . \tag{12.1}$$

Можна сформулювати критерій потрапляння роботи j в рішення S_j. Робота j включається в оптимальний розв'язок S_j тоді й тільки тоді, якщо:

$$v_j + OPT(p(j)) \geq OPT(j-1) \quad . \tag{12.2}$$

Це дає можливість сформулювати першу важливу складову динамічного програмування: рекурентне рівняння, яке виражає оптимальний розв'язок в термінах оптимальних розв'язків підзадач менших розмірностей.

З рівняння (12.1) можна побудувати рекурсивний алгоритм для обрахунку значення $OPT(n)$, припускаючи, що всі роботи вже були заздалегідь відсортовані за зростанням їх часу закінчення і визначенні значення $p(j)$ для всіх j.

```
ComputeOpt(j)
1  if j=0 then
2     return 0
3  else
4     return max(vⱼ + ComputeOpt(p(j)), CopmuteOpt(j-1))
```

Лістинг 12.1. Рекурсивний обрахунок значення $OPT(j)$.

Теорема 12.1. Процедура ComputeOpt коректно обраховує значення $OPT(j)$ для всіх j = 1...n.

Доведення. За означенням $OPT(0) = 0$. Візьмемо довільне $j > 0$ і припустимо за індуцією, що

ComputeOpt коректно обраховує значення *OPT*(i) для всіх $i < j$. За припущенням індукції ми знаємо, що ComputeOpt(*p*(*j*)) = *OPT*(*p*(*j*)) та ComputeOpt(*j*−1) = *OPT*(*j*−1), тож з (12.1) слідує, що

$$OPT(j) = \max(v_j + \text{ComputeOpt}(p(j)), \text{ComputeOpt}(j-1)) = \text{ComputeOpt}(j). \quad \blacksquare$$

На жаль, якщо реалізувати алгоритм ComputeOpt в представленому вигляді, то його час роботи буде експоненціальним у найгіршому випадку. Наприклад, на рис. 12.3 представлено дерево викликів для екземпляру задачі з рис. 12.2.

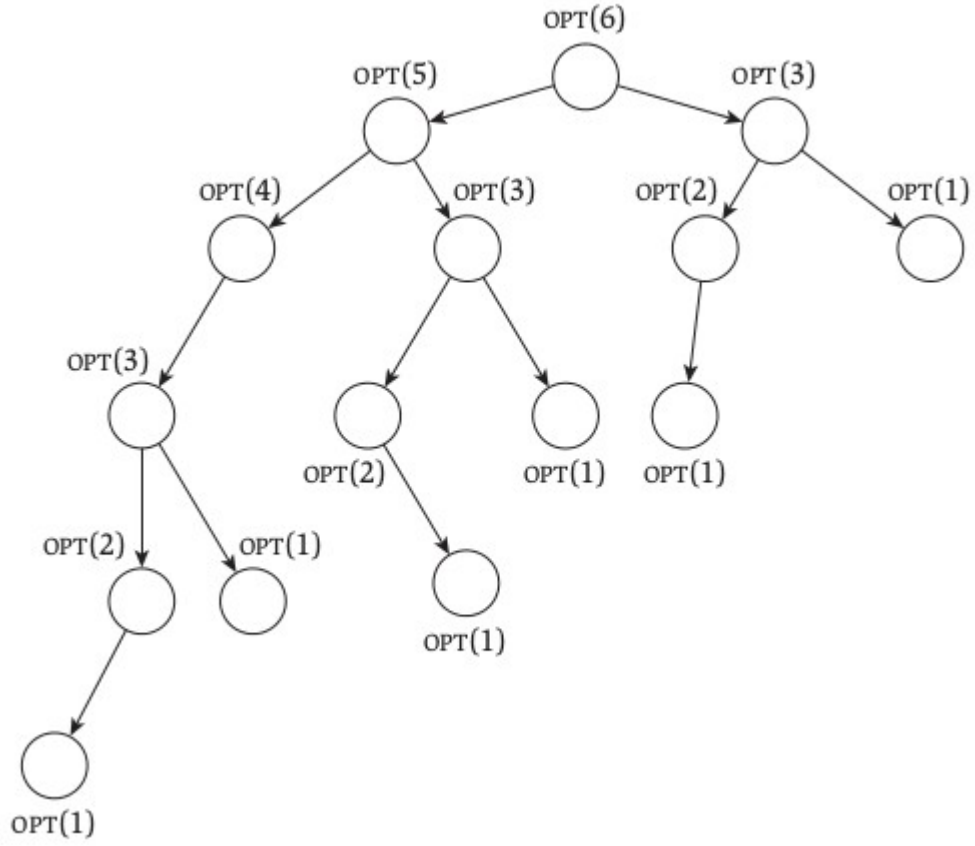


Рис. 12.3. Дерево підзадач під час роботи процедури ComputeOpt для екземпляру задачі з рис. 12.2.

Тож, аби позбутись проблеми експоненціального зростання, перейдемо до другого важливого принципу динамічного програмування. При детальному вивчення процедури ComputeOpt можна прийти до висновку, що вона розв’язує тільки $n + 1$ різних підзадач: ComputeOpt(0), ComputeOpt(1), ... , ComputeOpt(*n*). Те, що процедура працює експоненціально довго, слідує з повторних викликів для одних й тих самих екземплярів задачі.

Таким чином, для уникнення цієї зайвої роботи можна зберігати вже обраховані значення ComputeOpt у деякому глобальному просторі пам’яті, а потім звертатись до цих збережених даних в ті моменти, коли необхідно повторно обрахувати вже відоме рішення. Ця техніка збереження значень, які вже обраховані, називається мемоізація (memoization).

Реалізуємо стратегію мемоізації в більш інтелектуальній версії процедури ComputeOptM. Ця процедура буде використовувати масив *M*[0 ... *n*]; *M*[*j*] ініціалізується порожнім значенням, але після виклику ComputeOpt(*j*) буде зберігати його результат. Щоб обрахувати значення *OPT*(*n*), необхідно викликати ComputeOptM(*n*).

```
ComputeOptM(j)
1  if j=0 then
```

```
2     return 0
3  else if M[j] не порожнє then
4     return M[j]
5  else
6     M[j] = max(vⱼ + ComputeOptM(p(j)), CopmuteOptM(j-1))
7     return M[j]
```

Лістинг 12.2. Рекурсивний обрахунок значення *OPT*(*j*) із використанням мемоізації.

Теорема 12.2. Час роботи процедури ComputeOptM(*n*) становить O(*n*).

Доведення. Час, який витрачається на один виклик процедури ComputeOptM, не враховуючи рекурсивні виклики, є сталим — O(1). Тож загальний час роботи обмежується кількістю викликів процедури ComputeOptM. Розглянемо кількість комірок масиву *M*, які не є порожніми в довільний момент часу. Спочатку всі комірки масиву порожні, тож це значення дорівнює 0. Але кожного разу, коли відбувається рекурсивний виклик процедури ComputeOptM (рядок 6), нова комірка масиву *M* заповнюється отриманим значенням, а отже кількість непорожніх комірок зменшується на 1. Таким чином, загальна кількість рекурсивних викликів буде дорівнювати розмірності масиву *M*, тобто O(*n*). Це і буде визначати час роботи процедури ComputeOptM(*n*). ■

Наразі ми обрахували тільки саме значення оптимального розв'язку, а не сам розв'язок. Часто потрібно, щоб процедура повернули й сам розв'язок — підмножину інтервальних робіт в нашому випадку. Для цього можна додати кілька рядків у псевдокод процедури ComputeOptM, щоб на кожному етапі зберігати розв'язок поточної підзадачі. Але можна піти іншим шляхом і відновити розв'язок задачі за допомогою значень, які зберігаються в масиві *M*.

Ми знаємо з (12.2), що *j* належить оптимальному розв'язку підзадачі {1, ..., *j*} тоді й тільки тоді, коли $v_j + OPT(p(j)) \geq OPT(j-1)$. Використовуючи це перейдемо до наступної простої процедури, яка проходить від кінця до початку масиву *M* та визначає множину інтервальних робіт, які входять в оптимальний розв'язок.

```
FindSolution(j)
1  if j=0 then
2     return NULL
3  else
4     if vⱼ + M[p(j)] ≥ M[j-1] then
5        return { j } + FindSolution(p(j))
6     else
7        return FindSolution(j-1)
```

Лістинг 12.3. Визначення оптимального розв'язку за оптимальним значенням.

Час роботи процедури залежить від довжини масиву *M*, адже процедура викликає себе рекурсивно максимум для кожного елементу цього масиву. Відтак її час роботи становить O(*n*).

12.2. Принципи динамічного програмування

На основі розглянутого алгоритму для задачі складання розкладу зважених інтервальних робіт можна вивести загальні принципи динамічного програмування. В попередньому розділі ми спочатку розробили рекурсивний алгоритм, який працював за експоненціальний час, а потім зробили його більш ефективним за рахунок введення глобального масиву *M*, який використовувався для збереження рішень вже розв'язаних підзадач. Щоб детальніше зрозуміти ідею динамічного програмування, розглянемо ще одне представлення даного алгоритму.

Як вже зазначалось, основним фактором успіху для побудови ефективного алгоритму виявилась наявність масиву *M*: як тільки даний масив заповнений, задача є розв'язаною. Тож замість рекурсивного обрахунку значень цього масиву, можна обрахувати його напряму.

Починаючи з $M[0] = 0$ та продовжуючи збільшувати j, значення $M[j]$ визначається за (12.1). Новий алгоритм виглядає наступним чином.

```
IterativeComputeOpt( )
1  M[0] = 0
2  for j = 1 to n do:
3     M[j] = max(v_j + M[p(j)], M[j-1])
```

Лістинг 12.4. Ітеративний обрахунок розв'язків підзадач.

Аналогічно до теореми 12.1 ми можемо довести індукцією за j, що цей алгоритм записує значення $OPT(j)$ в масив значення $M[j]$. Так само можна передати обрахований масив M процедурі FindSolution для отримання оптимального рішення за його значенням. Очевидно, час роботи процедури IterativeComputeOpt становить $O(n)$, адже вона явно містить n ітерацій і витрачає сталий час на кожну ітерацію.

Робота процедури IterativeComputeOpt візуально представлена на рис. 12.4. На кожній ітерації алгоритм заповнює додаткову комірку масиву M, порівнюючи значення $v_j + M[p(j)]$ із $M[j-1]$.

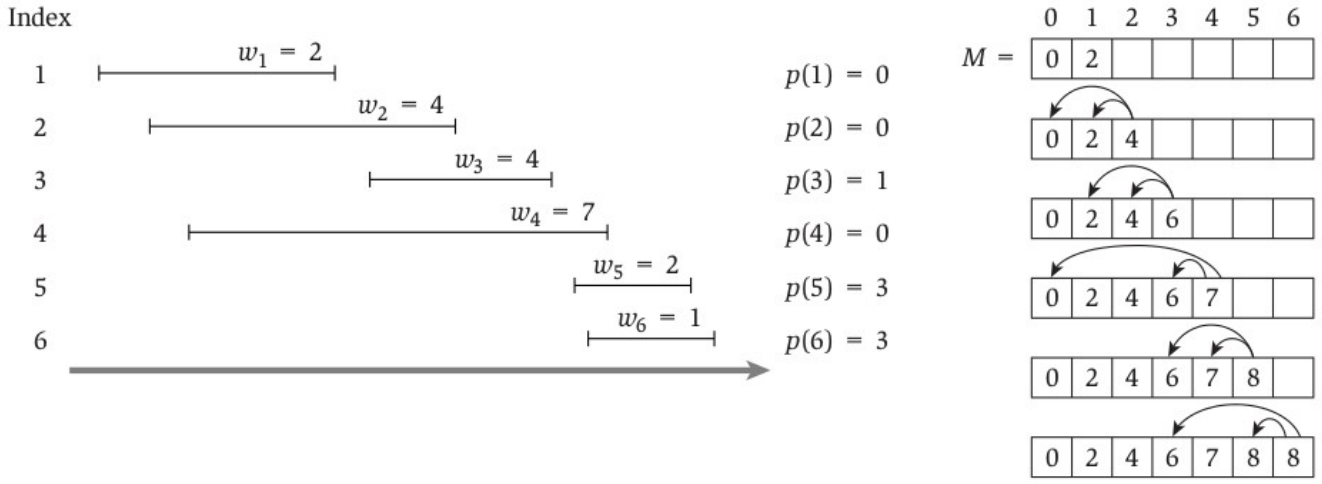


Рис. 12.4. Демонстрація роботи процедури IterativeComputeOpt.

Другий алгоритм для задачі складання розкладу зважених інтервальних робіт має багато спільного з першим, адже обидва мають витоки з рекурентного рівняння (12.1). З огляду на ці два алгоритми ми можемо вказати загальні правила, якими потрібно керуватись при розробці алгоритмів за методом динамічного програмування. В першу чергу мова йде про розбиття задачі на підзадачі, які задовольняють наступним властивостям:

1. Загальна кількість підзадач поліноміальна від розмірності вхідної задачі.
2. Розв'язок початкової задачі може бути легко отриманий через розв'язки окремих підзадач.
3. На підзадачах можна встановити природній порядок від "менших" до "більших" та рекурентне правило (на кшталт (12.1)), яке дозволяє отримувати розв'язок для більших підзадач через менші підзадачі.

Інколи легше почати розробку алгоритму динамічного програмування для конкретної задачі з формулювання набору підзадач, а потім визначати рекурентні зв'язки між ними. Проте частіше (як у прикладі з задачею складання розкладу зважених інтервальних робіт) має сенс визначити рекурентне співвідношення на основі структури оптимального рішення і після того визначати

необхідні для цього підзадачі. Ця двозначність, схожа на парадокс “курки та яйця”, є основною проблемою у фундаменті динамічного програмування.

12.3. Задача пошуку підмножин сум

Розглянемо наступну задачу складання розкладів, коли є один пристрій, на якому можуть запускатись завдання (роботи), а також множина робіт $\{1, 2, ..., n\}$. Даний ресурс може використовуватись лише в проміжку часу від 0 до деякого W. Кожна робота потребує час виконання w_i. Ціль полягає в тому, щоби обрати підмножину робіт S, які будуть максимально завантажувати ресурс до моменту W. Ця задача також має назву задача пошуку підмножин сум (*Subset Sum Problem*).

Ця задача є частковим випадком більш загальної задачі, яка називається задачею про рюкзак (*Knapsack Problem*), де кожний елемент (робота, запит) має два значення: цінність v_i та вагу w_i. Метою цієї задачі є вибір підмножини елементів, яка має максимальну цінність, так щоб загальна вага всіх елементів підмножини не перевищувала ліміту W. Задача про рюкзак часто виникає в інших більш складних проблемах. Назва цієї задачі походить від аналогії із заповненням фізичного рюкзака або валізи об’ємом W якомога більшою кількістю речей, так щоби їх загальна цінність була найбільшою.

Чи існує для цієї задачі коректний жадібний алгоритм? На жаль, відповідь не це питання негативна — наразі такий алгоритм не відомий. Природним жадібним підходом для даної задачі було би впорядкування робіт за спаданням ваг і потім відбір робіт до тих пір, поки не буде досягнута межа W. Але можна підібрати такий набір робіт з вагами $\{W/2 + 1, W/2, W/2\}$, для якого описаний жадібний підхід не поверне оптимальний розв’язок. З іншого боку, сортування у зростаючому порядку також не спрацює на наборі задач $\{1, W/2, W/2\}$.

Згадаємо основні принципи динамічного програмування: необхідно розбити початкову задачу на підзадачі меншої розмірності, з яких легко потім можна синтезувати розв’язок оригінальної задачі. Головна проблема тут — це визначення ефективної множини підзадач.

Спочатку спробуємо піти шляхом подібним до того, який було використано для задачі складання розкладу інтервальних робіт. Використаємо позначення $OPT(i)$, аналогічно до попередньої задачі, щоб позначити оптимальний розв’язок для підмножини робіт $\{1, ..., i\}$. Ключем для рішення тієї задачі був розгляд двох випадків, коли остання робота включалась в оптимальний розв’язок S чи ні. Так само і в новій задачі: якщо остання робота не включена в оптимальний розв’язок ($n \notin S$), то $OPT(n) = OPT(n-1)$.

Тепер необхідно розглянути випадок, коли $n \in S$. У випадку задачі складання розкладу інтервальних робіт, ми видаляли з поля зору всі задачі, які конфліктують з n. В новій задачі пошуку підмножин сум все не так просто: включення роботи n в оптимальний розв’язок не означає автоматичне виключення якихось інших робіт з розв’язку. Замість цього для набору робіт, які лишились, $\{1, ..., n-1\}$ в нас просто лишається менше вільного часу (простору): $W - w_n$ (рис. 12.5).

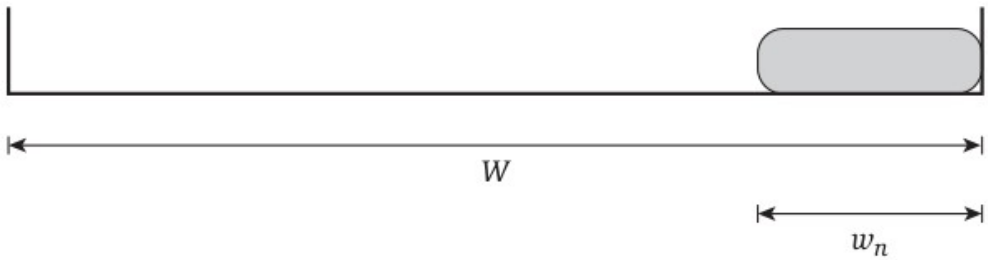


Рис. 12.5. Після того, як робота n включена у розв’язок, загальний об’єм W буде зменшено на w_n.

Це приводить нас до того, що для отримання вірного значення $OPT(n)$ окрім значення

OPT(*n*−1) необхідно ще знати який кращий розв'язок для перших *n*−1 робіт та допустимої ваги $W - w_n$. Тож ми будемо спиратись на більшу кількість підзадач: по одній для кожну підмножину {1, ..., *i*} та кожне доступне значення ліміту суми ваг. Припустимо, що *W* — ціле число і всі роботи мають цілі ваги w_i. Будемо розглядати підзадачу для кожного $i = 0, 1, ..., n$ та кожного цілого числа $0 \le w \le W$. Позначимо через *OPT*(*i*, *w*) значення оптимального розв'язку для підмножини робіт {1, ..., *i*} та максимальної дозволеної ваги *w*:

$$OPT(i, w) = \max_{U} \sum_{j \in U} w_j \ ,$$

де максимум береться по всіх підмножинах $U \subseteq \{1, ..., i\}$, які задовольняють умові $\sum_{j \in S} w_j \le w$. Використовуючи цей новий набір підзадач ми зможемо виразити значення *OPT*(*i*, *w*) через підзадачі меншої розмірності. Розв'язок оригінальної задачі буде визначатись величиною *OPT*(*n*, *W*). Як і раніше, нехай *S* буде позначати оптимальний розв'язок ориігнальної задачі.

- Якщо $n \notin S$, то *OPT*(*n*, *W*) = *OPT*(*n*−1, *W*), адже ми можемо просто проігнорувати роботу *n*.
- Якщо $n \in S$, то *OPT*(*n*, *W*) = w_n + *OPT*(*n*−1, *W*−w_n), адже доступний об'єм зменшується на значення ваги елементу *n*.

Якщо елемент *n* надто великий (*W* < w_n), тоді *OPT*(*n*, *W*) = *OPT*(*n*−1, *W*). Розглянувши деякий проміжний індекс *i*, отримаємо наступне рекурентне рівняння:

Якщо *w* < w_i, то *OPT*(*i*, *w*) = *OPT*(*i*−1, *w*). (12.3)

Інакше *OPT*(*i*, *w*) = max(*OPT*(*i*−1, *w*), w_i + *OPT*(*i*−1, *w*−w_i))

Ми представимо алгоритм, який заповнює всі значення *OPT*(*i*, *w*).

```
SubsetSum(n, W)
1  Визначити масив M[0...n,0...W]
2  for w = 0 to W do:
3     M[0,w] = 0
4  for i = 1 to n do:
5     for w = 0 to W do:
6        Використати рівняння (12.3) для обрахунку M[i,w]
7  return M[n,W]
```

Лістинг 12.5. Алгоритм розв'язання задачі пошуку підмножини сум

Використовуючи рівняння (12.3) за індукцією легко довести, що даний алгоритм коректно обраховує значення *M*[*n*, *W*] як оптимальний розв'язок задачі для *n* робіт та обмеження за сумарною вагою *W*.

Наприклад, розглянемо роботу алгоритму для екземпляра задачі, коли $W = 6$ і $n = 3$: $w_1 = w_2 = 2$ та $w_3 = 3$. Оптимальне значення буде дорівнювати *OPT*(3, 6) = 5. На рис. 12.6 представлені ітерації роботи описаного вище алгоритму при заповненні по рядках двомірної таблиці значень *OPT*.

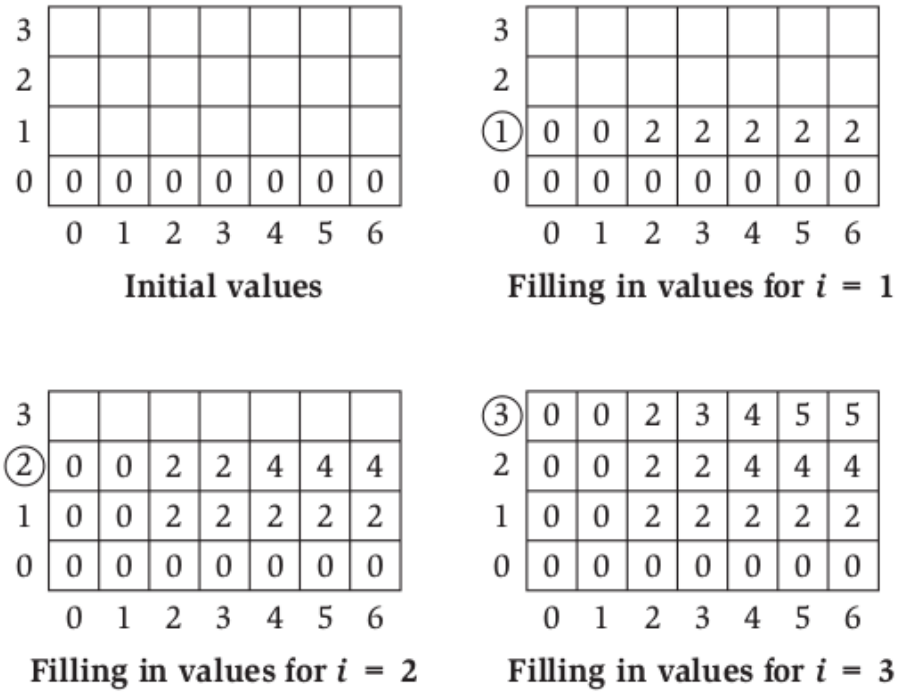


Рис. 12.6. Ітерації роботи процедури SubsetSum для $W = 6$ і $n = 3$: $w_1 = w_2 = 2$ та $w_3 = 3$.

Визначимо тепер час роботи процедури SubsetSum. Кожна ітерація процедури займає сталий час O(1) і за кожну ітерацію заповнюється одна комірка двовимірного масиву M, розмірність якого $W \times n$. Отже, час роботи процедури буде складати $O(nW)$.

Визначений час не є поліноміальним від n. Насправді він описується поліноміальною функцією від двох параметрів: n та W. Такі алгоритм називаються псевдо-поліноміальними. Псевдо-поліноміальні алгоритми можуть бути достатньо ефективними, якщо значення w_i достатньо малі; проте у випадку великих вхідних значень вони стають менш ефективними.

Щоб відновити оптимальну множину S робіт, можна пройти назад за таблицею M, як це робилось в попередньому розділі для задачі складання розкладу інтервальних робіт. Для заданої таблиці M оптимальних значень підзадач, оптимальну підмножину S можна визначити за час $O(n)$.

12.4. Задача про рюкзак

Задача про рюкзак є дещо складнішою за попередню. Розглянемо ситуацію, коли кожний елемент i має невід'ємну вагу w_i разом із цінніснім значенням v_i. Наша задача полягає в пошуку такої підмножини S, яка має максимальну сумарну цінність елементів, що входять в неї $\sum_{i \in S} v_i$, при обмеженні загального об'єму $\sum_{i \in S} w_i \le W$.

Неважко узагальнити попередній алгоритм динамічного програмування для даної задачі про рюкзак. Використаємо аналогічну множину підзадач $OPT(i, w)$ для позначення оптимального розв'язку при використанні множини елементів $\{1, ..., i\}$ та максимальної доступної ваги w. Позначимо оптимальний розв'язок через S. Тоді

- Якщо $n \notin S$, то $OPT(n, W) = OPT(n-1, W)$.
- Якщо $n \in S$, то $OPT(n, W) = v_n + OPT(n-1, W-w_n)$.

Використовуючи цю рекурсію ми можемо описати повністю аналогічний алгоритм динамічного програмування, який буде працювати також за час $O(nW)$.

12.5. Вирівнювання послідовностей

Онлайн словники стають все популярнішими і часто вони надають можливості, які не можна отримати в друкованих словниках: якщо ввести слово, яке відсутнє в словнику, то користувачу буде запропоновано слово (або декілька слів), яке, можливо, він мав на увазі, проте припустився помилки при введенні. Наприклад, для "*елктрастанція*" це буде "*електростанція*". Це визначається за допомогою близькості слів.

Інтуїтивно ми можемо сказати, що "*елктрастанція*" та "*електростанція*" схожі, тому що ми можемо зробити ці два слова однаковими, якщо додати літеру *е* до першого слова та замінити перше *а* на *о*. Обидві ці операції не виглядають складними, тож ми приходимо до висновку, що два слова є досить подібними. По-іншому, ми можемо записати два слова поруч:

```
ел-ктрастанція
електростанція
```

Дефіс (-) вказує на пропуск, де потрібно додати літеру в перше слово, щоб вирівняти його з другим. Але наше вирівнювання не є ідеальним, адже літера *а* в першому слові зіставлена з *о* в другому.

Тож, нам необхідна модель, в якій подібність слів визначається кількістю пропусків та неправильних літер під час вирівнювання двох слів. Звісно, існує багато варіантів як вирівняти два слова, наприклад:

```
ел-ктр---астанція
електроста---нція
```

В цьому випадку немає неправильних літер, але є аж 7 пропусків. Тож постає питання: що краще — наявність неправильних літер чи пропусків?

Схожа проблема — вирівнювання послідовностей літер — виникає в молекулярній біології. Геном будь-якого організму представлений довжелезними молекулами ДНК, відомими як хромосоми. Кожну хромосому можна легко представити як довгу стрічку, яка містить літери з алфавіту {A, C, G, T}. Рядки символів кодують інструкції з побудови білкових молекул; використовуючи хімічні механізми для зчитування елементів хромосоми, клітина може конструювати білки, які, в свою чергу, будуть контролювати її метаболізм.

Навіщо тут розглядати схожість між символьними рядками? Як вже було зазначено, послідовність символів в геномі можна розглядати як спосіб визначення властивостей організму. Наприклад, нехай ми маємо дві бактерії *X* та *Y*, які еволюційно тісно пов'язані між собою. Припустимо також, що ми визначили послідовність в ДНК для бактерії *X*, яка визначає деякий тип токсину. Якщо тепер ми визначимо схожу послідовність в ДНК для *Y*, то можна буде висунути припущення, що ця послідовність в *Y* також відповідає за той самий тип токсину. Використання подібних обчислювальних механізмів в біологічних експериментах є однією з найважливіших складових обчислювальної біології.

На початку 1970-х років два молекулярних біологи Нідлман (Needleman) та Уанщ (Wunsch) запропонували визначення подібності, яке з часом стало стандартним і сьогодні використовується скрізь. Це стало можливим завдяки простому та інтуїтивному підходу, на якому ґрунтувались ідеї згаданих науковців. Також це визначення подібності легко інтегрувалось в логіку динамічного програмування. Таким чином, парадигма динамічного програмування була повторно винайдена біологами через 20 років після того, як математики та інформатики її вперше висловили.

Це означення засноване на ідеях, які ми описали вище і пов'язано з "вирівнюванням" двох символьних рядків. Нехай задані два рядки *X* та *Y*, де *X* містить послідовність символів $x_1, x_2, ..., x_m$ та *Y* складається з символів $y_1, y_2, ..., y_n$. Розглянемо множини $\{1,2, ..., m\}$ та $\{1,2, ..., n\}$, які представляють різні позиції в рядках *X* та *Y*. Згадаємо, що результатом зіставлення двох множин є третя множина пар, така що кожен елемент зустрічається не більше ніж в одній парі. Будемо говорити, що зіставлення *M* двох множин — це вирівнювання, якщо в

ньому не присутні пари, що перетинаються: якщо (i, j), $(i', j') \in M$ та $i < i'$, тоді $j < j'$. Тож, вирівнювання дає спосіб зіставлення двох символьних рядків шляхом визначення пар позицій, які будуть знаходитись одна навпроти іншої. Наприклад:

```
stop-
-tops
```

відповідає вирівнювання {(2, 1), (3, 2), (4, 3)}.

Означення подібності буде базуватись на пошуку оптимального вирівнювання між X та Y відповідно до наступного критерію. Припустимо, що M — задане вирівнювання між X та Y.

- По-перше, існує параметр $\delta > 0$, який визначає штраф за пропуск. Для кожної позиції в X або Y, яка не увійшла до M, тобто є пропуском, нараховується штраф δ.
- По-друге, для кожної пари літер p, q в нашому алфавіті задається штраф за помилку (коли літери різні) — α_{pq}. Загалом припускається, що $\alpha_{pq} = 0$, якщо $p = q$.
- Вартість для M — це сума штрафів за пропуски та неправильні літери. Ціль полягає в мінімізації цього сумарного штрафу.

Процес мінімізації цієї вартості часто називається в біологічній літературі *вирівнюванням послідовностей*. Величини δ та $\{\alpha_{pq}\}$ є зовнішніми параметрами і визначаються розробником алгоритму; насправді, багато роботи приділяється якраз вибору правильних значень для цих параметрів. З нашої точки зору ми будемо вважати, що значення цих параметрів задані наперед.

Тож, зараз ми маємо конкретне числове означення подібності між рядками X та Y: це мінімальна вартість вирівнювання між X та Y. Чим нижча ця вартість, тим більш подібними є два символьні рядки. Тепер ми повернемось до задачі обрахунку цієї мінімальної вартості та оптимального вирівнювання для заданої пари рядків X та Y.

Один зі способів, яким можна вирішити цю задачу, — це динамічне програмування. І ми можемо зазначити наступне:

- В оптимальному вирівнюванні M або $(m,n) \in M$, або $(m,n) \notin M$. Тобто або два останні символи обох рядків зіставлені між собою, або ні.

Сам по собі цей факт є надто слабким для розробки розв'язку методом динамічного програмування. Але ми можемо підсилити його наступною теоремою.

Теорема 12.3. Нехай M — це вирівнювання для X та Y. Якщо $(m,n) \notin M$, тоді або позиція m в X, або позиція n в Y не буде зіставлена в M взагалі.

Доведення. Припустимо, що $(m,n) \notin M$, але є такі числа $i < m$ та $j < n$, що $(m, j) \in M$ та $(i,n) \in M$. Але це суперечить нашому означенню вирівнювання: маємо $(i,n) \in M$, $(m, j) \in M$ з $i < m$, але $j < n$, тож пари (i, n) та (m, j) перетинаються. ■

Існує еквівалентний запис теореми 12.3, який представляє три альтернативних варіанти. Це нас приводить до рекурентного формулювання.

В оптимальному вирівнюванні M виконується хоча би одне з наступних тверджень: (12.4)

1) $(m,n) \in M$;
2) позиція m в X не зіставляється з жодною позицію в Y;
3) позиція n в Y не зіставляється з жодною позицію в X.

Тепер припустимо, що $OPT(i, j)$ позначає вартість мінімального вирівнювання між $x_1, x_2, ..., x_i$ та $y_1, y_2, ..., y_j$. У випадку (1) в (12.4) ми будемо платити штраф α_{x_m, y_n} і потім переходити до вирівнювання $x_1, x_2, ..., x_{m-1}$ та $y_1, y_2, ..., y_{n-1}$: $OPT(m,n) = \alpha_{x_m, y_n} + OPT(m-1, n-1)$. У випадку (2) ми платимо штраф δ за пропуск, адже позиція m в X не зіставляється з жодною позицію в Y; і після того переходимо до вирівнювання $x_1, x_2, ..., x_{m-1}$ та $y_1, y_2, ..., y_n$: $OPT(m,n) = \delta + OPT(m-1, n)$. Аналогічно у випадку (3) отримуємо $OPT(m,n) = \delta + OPT(m, n-1)$. Звідси отримуємо рекурентне співвідношення:

$$OPT(i, j) = \min\{\alpha_{x_i, y_j} + OPT(i-1, j-1),\ \delta + OPT(i-1, j),\ \delta + OPT(i, j-1)\} \quad (12.5)$$

Отже, ми отримали рекурентне співвідношення для побудови алгоритму динамічного

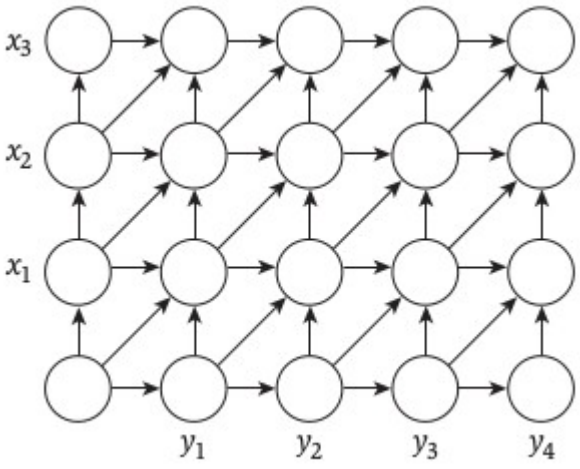
програмування. Загалом існує O(*mn*) підзадач і кінцевою відповіддю на поставлену задачу буде значення *OPT*(*m*, *n*). Тепер можна описати алгоритм для визначення оптимального вирівнювання. Для визначення початкових значень скористаємось тим, що *OPT*(*i*, 0) = *OPT* (0, *i*) = *iδ* для всіх *i*, адже єдиний спосіб зіставити *i*-у літеру з нульовою — це пропуск.

```
Alignment(X, Y)
1  Визначити масив A[0...m, 0...n]
2  for i = 0 to m do:
3     A[i,0] = iδ
4  for j = 0 to n do:
5     A[0,j] = jδ
6  for j = 1 to n do:
7     for i = 1 to m do:
8        Використати рівняння (12.5) для обрахунку A[i,j]
9  return A[m,n]
```

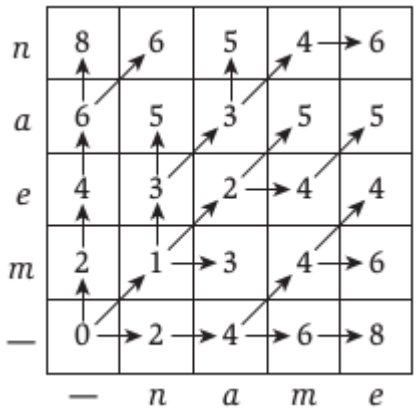
Лістинг 12.6. Алгоритм розв’язання задачі вирівнювання послідовностей

Як і в попередньому алгоритмі, ми можемо пройти масивом з кінця в початок для відновлення самого рішення вирівнювання. Час роботи алгоритму визначається кількістю елементів в масиві *A* і становить O(*mn*).

Зазначений алгоритм розв’язання задачі вирівнювання послідовностей можна зручно представити графічно. Побудуємо двовимірну ґратку G_{XY} розмірності *m*×*n*, рядки якої позначені літерами з *X*, а стовпці — літерами з *Y* (рис 12.7,*а*). Пронумеруємо рядки від 0 до *m*, а стовпці від — від 0 до *n*. Позначимо вузол у рядку *i* та стовпці *j* через (*i*, *j*). Введемо вартість дуг графу G_{XY}: варість кожної горизонтальної або вертикальної дуги становить δ і вартість діагональної дуги з (*i*−1, *j* − 1) до (*i*, *j*) становить α_{x_i,y_j} .



а



б

Рис. 12.7. (*а*) Представлення вирівнювання послідовностей за допомогою графу. (*б*) Значення *OPT* для задачі вирівнювання слів *mean* та *name*.

Тепер можна представити рівняння (12.5) через пошук шляху мінімальної вартості в графі G_{XY} з (0, 0) до (*i*, *j*).

Теорема 12.4. Позначимо через *f*(*i*, *j*) вартість мінімального шляху з (0, 0) до (*i*, *j*) в графі G_{XY}. Тоді для всіх *i*, *j* *f*(*i*, *j*) = *OPT*(*i*, *j*).

Доведення. Дану теорему можна легко довести за індукцією по *i* + *j*. Коли *i* + *j* = 0, *i* = *j* = 0 і

очевидно, що $f(i, j) = OPT(i, j) = 0$.

Розглянемо довільні значення для i та j та припустимо, що твердження вірно для всіх пар (i', j'), таких що $i' + j' < i + j$. Остання дуга найкоротшого шляху до (i, j) йде або з $(i\text{-}1, j\text{-}1)$, або з $(i\text{-}1, j)$, або з $(i, j\text{-}1)$. Тож маємо:

$$f(i, j) = \min\{\alpha_{x_i, y_j} + f(i-1, j-1),\ \delta + f(i-1, j),\ \delta + f(i-1, j)\}$$
$$= \min\{\alpha_{x_i, y_j} + OPT(i-1, j-1),\ \delta + OPT(i-1, j),\ \delta + OPT(i-1, j)\} = OPT(i, j),$$

де ми переходимо від першої рівності до другої за припущенням індукції, а від другої до третьої — за (12.5). ■

Тож, значення оптимального вирівнювання дорівнює довжині найкоротшого шляху в G_{XY} з (0, 0) до (m, n). Більше того, діагональні дуги, які включаються в цей найкоротший шлях, відповідають парам символів, які входять до оптимального вирівнювання. Наприклад, на рис. 12.7,*б* представлено найкоротший шлях з (0, 0) до кожного вузла (i, j) для проблеми вирівнювання двох слів *mean* та *name*. Тут ми використали значення $\delta = 2$; невірне зіставлення голосної з іншою голосною або приголосної з іншою приголосною — штраф 1; зіставлення голосної з приголосною — штраф 3. Для кожної комірки таблиці стрілка представляє останній перехід в найкоротшому шляху, який веде до відповідного вузла графу. Тож, рухаючись назад за стрілками від вузла (4, 4) можна відновити оптимальне вирівнювання.

Література

J. Kleinberg, E. Tardos. Algorithm Design. Addison Wesley, 2005. Глава 6.