Курс kiev-clrs – Лекция 15. Динамическое программирование

Олег Смирнов

4 июля 2009 г.

Содержание

1 Цель лекции

- Динамическое программирование и его приложения
- Мемоизация

2 Введение

Слово “программирование” в названии метода не относится к программированию как к написанию программ, так же как и в “линейном программировании”. Исторически так называли табличные методы, как в случае перфокарт.

Динамическое программирование является методом дизайна алгоритмов, так же как “разделяй и властвуй”, “жадные алгоритмы”, “рандомизация” и т.д.

3 Наибольшая общая подпоследовательность

Хорошим примером применения метода является задача поиска наибольшей общей подпоследовательности (Longest Common Subsequence, LCS).

Для двух заданных строк (множест элементов) $x[1\,..\,m]$ и $y[1\,..\,n]$ найти наибольшие общие последовательности элементов (их может быть несколько).

Задача возникает, например, в вычислительное биологии, где требуется сравнивать цепочки ДНК.

В этом примере $LCS(x, y) = BCBA, BDAB, BCAB, ..$.

x	A	B	C	B	D	A	B
y	B	D	C	A	B	A	

В случае наивного алгоритма поиска LCS методом brute force: для каждой подпоследовательности из x проверить, содержится ли она в y. Тогда:

- время проверки каждой подпоследовательности в y: $O(n)$

- общее количество подпоследовательностей в x: 2^m (рассматривая вектор битов длинной m, каждый бит которого означает, включать ли соответствующий элемент)
- время работы алгоритма в худшем случае: $O(n2^m)$ – экспоненциально

Более оптимальный алгоритм поиска LCS состоит из двух шагов:

1. Вычисление *длины* наибольшей общей подпоследовательности
2. Поиск подпоследовательности заданной длины

Очевидно, что наибольшая длина, полученная на первом шаге, является искомой. Алгоритм становится проще за счёт того, что необходимо работать только с числами – длинами LCS. Обозначим длину последовательности s через $|s|$.

Важным шагом в дизайне алгоритмов динамического программирования является рассмотрения подзадач. Необходимо показать, что оптимальное решение исходной задачи содержит оптимальные решения подзадач меньшего размера.

Рассмотрим *префиксы* x и y.

Пусть $c[i,j] = |LCS(x[1\,..\,i], y[1\,..\,j])|$ – длина общей подпоследовательности для префиксов x и y длины i и j соответственно. Вычислив для всех i и j, получим $c[m,n] = |LCS(x,y)|$ – искомая наибольшая подпоследовательность.

Значение $c[i,j]$ можно вычислить рекурсивно. Докажем теорему:

$$c[i,j] = \begin{cases} c[i-1,j-1]+1, \text{ если } x[i] = y[j] \\ max\{c[i-1,j], c[i,j-1]\}, \text{ в противном случае} \end{cases}$$

Т.е. длина общей последовательности двух префиксов на единицу больше предыдущих префиксов, если их конечные элементы совпадают, или равна максимуму из двух длин для двух предыдущих префиксов в противном случае.

Рассмотрим случай $x[i] = y[j]$.

Пусть $z[1\,..\,k] = LCS(x[1\,..\,i], y[1\,..\,j])$ – LCS префиксов, тогда $c[i,j] = k$ – длина z. Тогда последний символ подпоследовательности должен

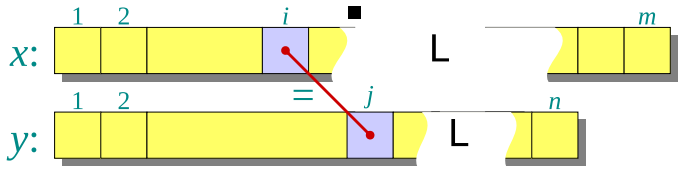

Рис. 1: Задача LCS

быть равен $z[k] = x[i](= y[j])$ иначе z можно было бы удлинить, добавив $x[i]$.

Следствие: $z[1\,..\,k-1]$ является наибольшей подпоследовательностью префиксов:

$$z[1\,..\,k-1] = LCS(x[1\,..\,i-1], y[1\,..\,j-1])$$

Пусть w – более длинная подпоследовательность $x[1\,..\,i-1]$ и $y[1\,..\,j-1]$, т.е. $|w| > k-1$. Тогда (аргумент "cut and paste") конкатенация $w \parallel z[k]$ также является общей подпоследовательностью $x[1\,..\,i]$, $y[1\,..\,j]$ длины $|w \parallel z[k]| > k$. Это противоречит доказывает следствие.

Таким образом $c[i-1, j-1] = k-1$, откуда следует, что $c[i,j] = c[i-1, j-1] + 1$. Остальные случаи симметричны.

Это свойство называется "оптимальной подструктурой" или первым признаком динамического программирования.

Первый признак динамического программирования: оптимальное решение исходной задачи содержит оптимальные решения подзадач меньшего размера.

В случаи задачи LCS это означает, что если $z = LCS(x, y)$, то любой префикс z является LCS префикса x и префикса y.

Для проверки первого признака обычно используют аргумент "cut and paste".

Испльзуя теорему, можно записать рекурсивный алгоритм:

LCS(x, y, i, j)
1 **if** $x[i] = y[i]$
2 **then** $c[i,j] \leftarrow LCS(x, y, i-1, i-1) + 1$
3 **else** $c[i,j] \leftarrow max\{LCS(x, y, i-1, j), LCS(x, y, i, j-1)\}$
4 **return** $c[i,j]$

В худшем случае, если $x[i] \neq y[j]$ алгоритм выполняет рекурсивное вычисление двух подзадач, по очереди декрементируя один из параметров. Дерево рекурсии алгоритма имеет высоту $m + n$, т.е. время выполнения

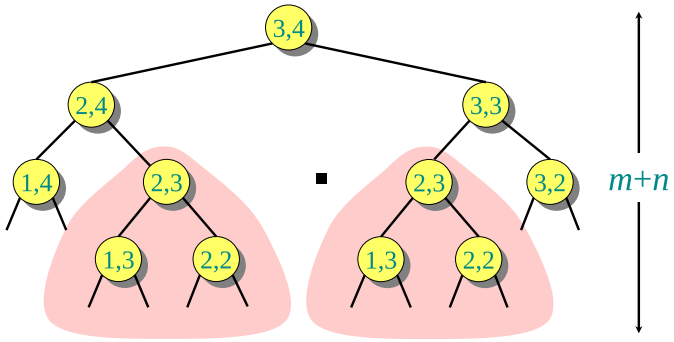

Рис. 2: Дерево рекурсии

по прежнему будет экспоненциально.

Можно заметить, что некоторые поддеревья в нём повторяются несколько раз. Вычисляя соответствующие подзадачи однократно, время работы можно сократить.

Это свойство называется “перекрывающимися подзадачами” или вторым признаком динамического программирования.

Второй признак динамического программирования: рекурсивное решение содержит небольшое количество различных подзадач, которые повторяются многократно.

Количество различных подзадач в LCS равно nm, что гораздо меньше $n2^m$.

4 Мемоизация

Для решения можно применить подход, называемый мемоизацией (memoization): после вычисления решения подзадачи оно записывается в таблицу и при повторном вызове уже не вычисляется, а берется из таблицы.

```
LCS(x, y, i, j)
1  if c[i, j] = NIL
2     then if x[i] = y[i]
3             then c[i, j] ← LCS(x, y, i − 1, i − 1) + 1
4             else  c[i, j] ← max{LCS(x, y, i − 1, j), LCS(x, y, i, j − 1)}
5  return c[i, j]
```

Амортизированное время работы этого алгоритма $T(m,n) = \Theta(mn)$. Объем используемой памяти $Space(m,n) = \Theta(mn)$.

Мемоизация хорошо подходит для функций, которые возвращают одинаковые значения для одинаковых входных параметров. Этот приём не работает для функций с побочными эффектами.

5 Динамическое программирование

Алгоритм с мемоизацией использует значительные объем памяти и выполняет вычисления в обратном порядке (сверху-вниз). Идея динамического программирования, заключается в том, что таблицу значений $c[i,j]$ можно вычислять последовательно, снизу-вверх:

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

Рис. 3: Таблица LCS

Порядок заполнения таблицы:

- первая строка таблицы заполняется нулями
- каждый элемент последовательно заполняется по формуле для $c[i, j]$: если символы для позиции i, j совпадают, то в неё записывается значение $c[i-1, j-1]+1$, иначе вычисляется максимум от соседей слева и сверху

Элемент в правом нижнем углу показывает длину наибольшей общей подпоследовательности. Для построения искомой последовательности нужно пройти по таблице в обратную сторону.

Порядок прохода по таблице:

- если элемент получен как максимум от двух префиксов – перейти в позицию с наибольшим префиксом
- если элемент получен прибавлением единицы к соседу по диагонали – перейти в эту позицию и добавить соответствующий символ к последовательности
- выбирая различные эквивалентные пути можно получить все возможные последовательности

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

Рис. 4: Таблица LCS

Алгоритм использует $\Theta(mn)$ памяти. Объем памяти можно сократить до $O(min\{m, n\})$, если хранить вместо всей таблицы только одну (предыдущую) строчку, которая необходима для вычисления элемента.

6 Задачи динамического програмимирования

Рассмотренный алгоритм LCS используется, в частности, в программе diff. Существует еще ряд известных задач, решаемых методом динамического программимирования.

6.1 Редакторское расстояние

Расстоянием Левенштейна (также редакторским расстояним, editor's distance) называется мера разницы двух последовательностей символов (строк) относительно минимального количества операций вставки, удаления и замены, необходимых для перевода одной строки в другую.

Задача поиска дистанции Левенштейна решается с помощью динамического программирования. Дистанция используется в алгоритма fuzzy matching (например, проверка правописания) и в текстовых редакторах.

6.2 Перемножение цепочки матриц

Число итераций, необходимое для умножения двух матриц размера $p \times q$ и $q \times r$ равно $p \cdot q \cdot r$. Умножение матриц ассоциативно, т.е. три матрицы A_1, A_2, A_3 размерности 10×100, 100×5 и 5×50 соответственно, можно перемножить двумя возможными способами: $((A_1 \cdot A_2) \cdot A_3)$ и $(A_1 \cdot (A_2 \cdot A_3))$. Однако в первом случае будет выполнено$10 \cdot 100 \cdot 5 + 10 \cdot 5 \cdot 50 = 7500$ итераций. А во втором – $100 \cdot 5 \cdot 50 + 10 \cdot 100 \cdot 50 = 75000$ итераций.

Можно показать, что число возможных вариантов перемножения нескольких матриц равно числу вариантов расстановки скобок между ними, которое равно числу Каталана $C_n \sim \frac{4^n}{n^{3/2}}$ – оно растёт экспоненциально.

С помощью динамического программирования, можно найти оптимальный порядок перемножения матриц за $O(n^3)$ итераций.

6.3 Поиск оптимального пути в графе

Задача поиска поиска оптимальных путей из одной вершины нагруженного графа в остальные решается алгоритмом Дейкстры (для графа без дуг отрицательного веса) и алгоритмом Беллмана-Форда (с ними).

При решении каждой вершине из графа V сопоставляется метка – минимальное известное расстояние от этой вершины до начальной a. Алгоритм работает пошагово – на каждом шаге он "посещает" одну вершину и пытается уменьшать метки. Работа завершается, когда все вершины посещены. В начале работы метки вершин принимаются равными бесконечности. На каждом шаге вершины просматриваются в порядке BFS (поиск в ширину).

Практическим приложением этого алгоритма является протокол сетевой маршрутизации OSPF, который вычисляет оптимальный путь между узлами сети по заданной таблице маршрутизации.

7 Заключительные замечания

Мемоизация очень удобна в строго функциональных языках программирования, благодаря отстутствию побочных эффектов и наличию ленивых вычислений.