Глава 6

Динамическое программирование

Наше изучение алгоритмических методов началось с жадных алгоритмов, которые в определенном смысле представляют наиболее естественный подход к разработке алгоритма. Как вы видели, столкнувшись с новой вычислительной задачей, иногда можно легко предложить для нее несколько возможных жадных алгоритмов; проблема в том, чтобы определить, дают ли какие-либо из этих алгоритмов верное решение задачи во всех случаях.

Все задачи из главы 4 объединял тот факт, что в конечном итоге действительно находился работающий жадный алгоритм. К сожалению, так бывает далеко не всегда; для большинства задач настоящие трудности возникают не с выбором правильной жадной стратегии из несколько вариантов, а с тем, что естественного жадного алгоритма для задачи вообще не существует. В таких случаях важно иметь наготове другие методы. Принцип «разделяй и властвуй» иногда выручает, однако реализации этого принципа, которые мы видели в предыдущей главе, часто недостаточно эффективны для сокращения экспоненциального поиска «грубой силой» до полиномиального времени. Как было замечено в главе 5, его применения чаще позволяют сократить излишне высокое, но уже полиномиальное время выполнения до более быстрого.

В этой главе мы обратимся к более мощному и нетривиальному методу разработки алгоритмов — *динамическому программированию*. Охарактеризовать динамическое программирование будет проще после того, как вы увидите его в действии, но основная идея базируется на интуитивных представлениях, лежащих в основе принципа «разделяй и властвуй», и по сути противоположна жадной стратегии: алгоритм неявно исследует пространство всех возможных решений, раскладывает задачу на серии *подзадач*, а затем строит правильные решения подзадач все большего размера. В некотором смысле динамическое программирование может рассматриваться как метод, работающий в опасной близости от границ перебора «грубой силой»: хотя оно систематически прорабатывает большой набор возможных решений задачи, это делается без явной проверки всех вариантов. Именно из-за этой необходимости тщательного выдерживания баланса динамическое программирование бывает трудно освоить; как правило, вы начинаете чувствовать себя уверенно только после накопления немалого практического опыта.

Помня об этом, мы рассмотрим первый пример динамического программирования: задачу взвешенного интервального планирования, определение которой приводилось в разделе 1.2. Разработка алгоритма динамического программирования для этой задачи будет проводиться в два этапа: сначала как рекурсивная процедура, которая сильно напоминает поиск «грубой силой», а затем, после переосмысления этой процедуры, — как итеративный алгоритм, который строит решения последовательно увеличивающихся подзадач.

6.1. Взвешенное интервальное планирование: рекурсивная процедура

Мы уже видели, что жадный алгоритм обеспечивает оптимальное решение задачи интервального планирования, целью которой является получение множества неперекрывающихся интервалов максимально возможного размера. Задача взвешенного интервального планирования имеет более общий характер: с каждым интервалом связывается определенное значение (*вес*), а решением задачи считается множество с максимальным суммарным весом.

Разработка рекурсивного алгоритма

Поскольку исходная задача интервального планирования представляет собой частный случай, в котором все веса равны 1, мы уже знаем, что большинство жадных алгоритмов не обеспечивает оптимального решения. Но даже алгоритм, который работал ранее (многократный выбор интервала, завершающегося раньше всех), в этой ситуации уже не является оптимальным, как показывает простой пример на рис. 6.1.

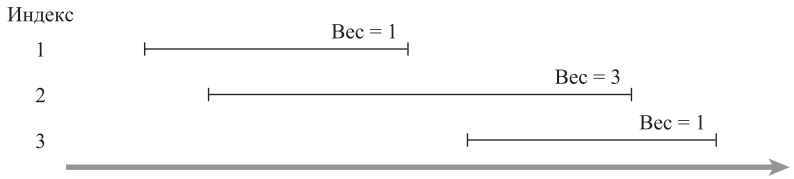

Рис. 6.1. Простой пример задачи взвешенного интервального планирования

Действительно, естественный жадный алгоритм для этой задачи неизвестен, что заставляет нас переключиться на динамическое программирование. Как упоминалось ранее, знакомство с динамическим программированием начнется с рекурсивного алгоритма, а затем в следующем разделе мы перейдем на итеративный метод, более близкий по духу к алгоритмам оставшейся части этой главы.

Воспользуемся определениями, приведенными в формулировке задачи интервального планирования из раздела 1.2. Имеются n заявок с пометками 1, ..., n; для

каждой заявки i указывается начальное время s_i и конечное время f_i. С каждым интервалом i связывается некоторое значение — вес v_i. Два интервала называются *совместимыми*, если они не перекрываются. Целью текущей задачи является нахождение подмножества $S \subseteq \{1, ..., n\}$ взаимно совместимых интервалов, максимизирующего сумму весов выбранных интервалов $\sum_{i \in S} v_i$.

Предположим, заявки отсортированы в порядке неубывания конечного времени: $f_1 \leq f_2 \leq ... \leq f_n$. Заявка i называется предшествующей заявке j, если $i < j$. Таким образом определяется естественный порядок «слева направо», в котором будут рассматриваться интервалы. Чтобы было удобнее обсуждать этот порядок, определим $p(j)$ для интервала j как наибольший индекс $i < j$, при котором интервалы i и j не перекрываются. Другими словами, i — крайний левый интервал, который завершается до начала j. Определим $p(j) = 0$, если не существует заявки $i < j$, не перекрывающейся с j. Пример определения $p(j)$ изображен на рис. 6.2.

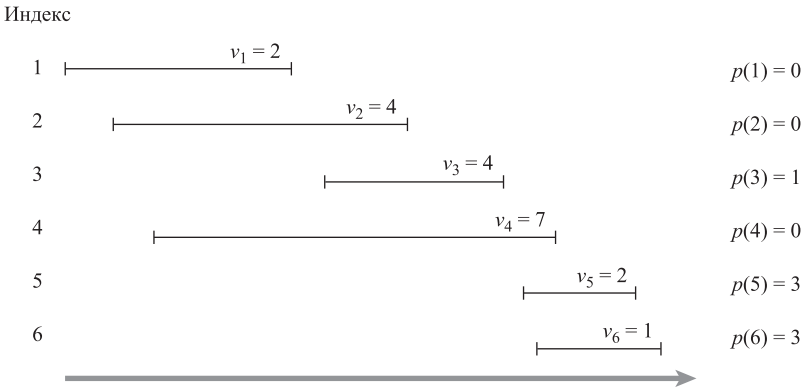

Рис. 6.2. Задача взвешенного интервального планирования с функциями $p(j)$, определенными для каждого интервала j

Допустим, у нас имеется экземпляр задачи взвешенного интервального планирования; рассмотрим оптимальное решение O, временно забыв о том, что нам о нем ничего не известно. Есть нечто совершенно очевидное, что можно утверждать о решении O: либо интервал n (последний) принадлежит O, либо нет. Попробуем исследовать обе стороны этой дихотомии. Если $n \in O$, то очевидно, ни один интервал, индексируемый строго между $p(n)$ и n, не может принадлежать O, потому что по определению $p(n)$ мы знаем, что интервалы $p(n) + 1, p(n) + 2, ..., n - 1$ — все они перекрывают интервал n. Более того, если $n \in O$, то решение O должно включать *оптимальное* решение задачи, состоящей из заявок $\{1, ..., p(n)\}$, потому что в противном случае выбор заявок O из $\{1, ..., p(n)\}$ можно было бы заменить лучшим выбором без риска перекрытия заявки n.

С другой стороны, если $n \notin O$, то O просто совпадает с оптимальным решением задачи, состоящей из заявок $\{1, ..., n - 1\}$. Рассуждения полностью аналогичны: предположим, что O не включает заявку n; если оно не выбирает оптимальное множество заявок из $\{1, ..., n - 1\}$, то его можно заменить лучшим выбором.

Все это наводит на мысль, что в процессе поиска оптимального решения для интервалов {1, 2, ..., n} будет производиться поиск оптимальных решений меньших задач в форме {1, 2, ..., j}. Пусть для каждого значения j от 1 до n оптимальное решение задачи, состоящей из заявок {1, ..., j}, обозначается O_j, а значение (суммарный вес) этого решения — OPT(j). (Также мы определим OPT(0) = 0 как оптимум по пустому множеству интервалов.) Искомое оптимальное решение представляет собой O_n со значением OPT(n). Для оптимального решения O_j по интервалам {1, 2, ..., j} из приведенных выше рассуждений (обобщенных для $j = n$) следует, что либо $j \in O_j$, и тогда OPT(j) = v_j + OPT($p(j)$), либо $j \notin O_j$, и тогда $OPT(j)$ = OPT($j - 1$). Так как других вариантов быть не может ($j \in O_j$ или $j \in O_j$), можно утверждать, что

(6.1) OPT(j) = max(v_j + OPT($p(j)$), OPT($j - 1$)).

Как решить, принадлежит ли n оптимальному решению O_j? Это тоже несложно: n принадлежит оптимальному решению в том и только в том случае, если первый из упомянутых выше вариантов по крайней мере не хуже второго; другими словами,

(6.2) Заявка j принадлежит оптимальному решению для множества {1, 2, ..., j} в том и только в том случае, если

$$v + \text{OPT}(p(j)) \geq \text{OPT}(j - 1).$$

Эти факты образуют первый важнейший компонент, на котором базируется решение методом динамического программирования: рекуррентное уравнение, которое выражает оптимальное решение (или его значение) в контексте оптимальных решений меньших подзадач.

Несмотря на простые рассуждения, которые привели нас к этому заключению, утверждение (6.1) само по себе является существенным достижением. Оно напрямую предоставляет рекурсивный алгоритм для вычисления OPT(n) при условии, что заявки уже отсортированы по времени завершения и для каждого j вычислены значения $p(j)$.

```
Compute-Opt(j)
   Если j = 0
      Вернуть 0
   Иначе
      Вернуть max(v_j + Compute-Opt(p(j)), Compute-Opt(j – 1))
   Конец Если
```

Правильность алгоритма напрямую доказывается индукцией по j:

(6.3) Compute-Opt(j) правильно вычисляет OPT(j) для всех j = 1, 2, ..., n.

Доказательство. По определению OPT(0) = 0. Теперь возьмем некоторое значение $j > 0$ и предположим, что Compute-Opt(i) правильно вычисляет OPT(i) для всех $i < j$. Из индукционной гипотезы следует, что Compute-Opt($p(j)$) = OPT($p(j)$) и Compute-Opt($j - 1$) = OPT($j - 1$); следовательно, из (6.1)

OPT(j) = max(v_j + Compute-Opt($p(j)$), Compute-Opt($j - 1$))
= Compute-Opt(j) ■

К сожалению, если реализовать алгоритм Compute-Opt в этом виде, его выполнение в худшем случае займет экспоненциальное время. Например, на рис. 6.3 изображено дерево вызовов для экземпляра задачи на рис. 6.2: оно очень быстро расширяется из-за рекурсивного ветвления. Или более экстремальный пример: в системе с частым наслоением наподобие изображенной на рис. 6.4, где $p(j) = j - 2$ для всех $j = 2, 3, 4, ..., n$, мы видим, что Compute-Opt(j) генерирует раздельные рекурсивные вызовы для задач с размерами $j - 1$ и $j - 2$. Другими словами, общее количество вызовов Compute-Opt в этой задаче будет расти в темпе чисел Фибоначчи, что означает экспоненциальный рост. Таким образом, решение с полиномиальным временем так и остается нереализованным.

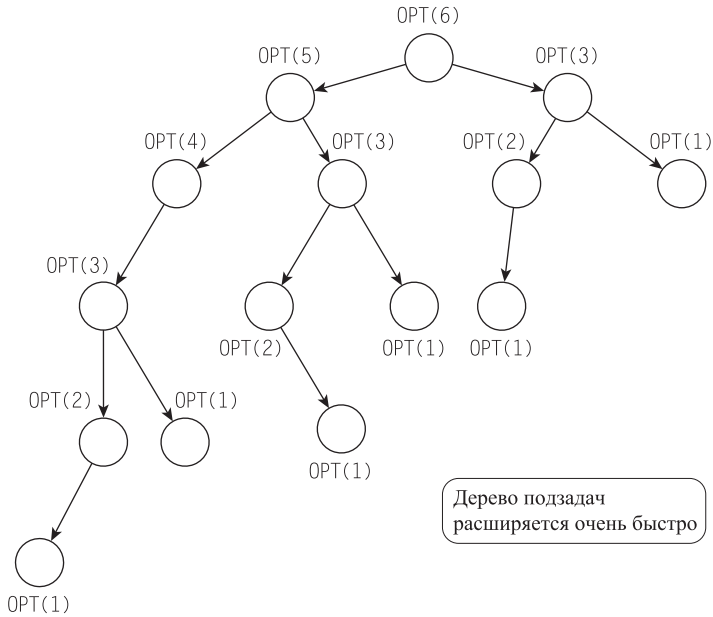

Рис. 6.3. Дерево подзадач, вызываемых Compute-Opt, для экземпляра задачи на рис. 6.2

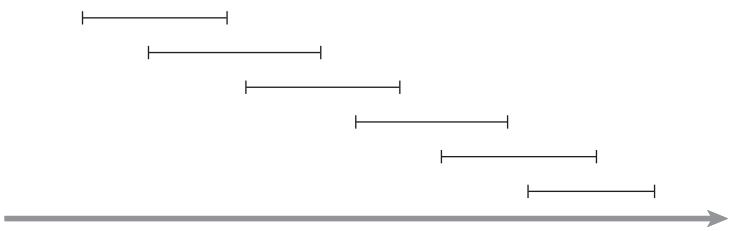

Рис. 6.4. Экземпляр задачи взвешенного интервального планирования, для которого простая рекурсия Compute-Opt занимает экспоненциальное время. Веса интервалов в этом примере равны 1