УДК 519.854.64

Романовский Иосиф Владимирович

ПЕРЕБОР СУБОПТИМАЛЬНЫХ РЕШЕНИЙ В ДИСКРЕТНЫХ ЗАДАЧАХ ОПТИМИЗАЦИИ

Аннотация

В статье рассматривается подход к перебору субоптимальных решений, основанный на введении специальных процессов перебора – «перечислителей», решающих задачи такого типа, и создании специальных, достаточно простых операций, позволяющих получать перечислители из более простых. Приводятся примеры таких операций.

Ключевые слова: субоптимальные решения, дискретные задачи оптимизации, динамическое программирование.

ЧТО ТАКОЕ СУБОПТИМАЛЬНЫЕ РЕШЕНИЯ И ЗАЧЕМ ОНИ НУЖНЫ

Эта статья посвящена задачам перебора *k-оптимальных* решений в задачах дискретной оптимизации, то есть решений, следующих за оптимальными по значению целевой функции в задачах оптимизации. Такие следующие решения принято называть *субоптимальными*.

Почему в дискретной оптимизации важен перебор субоптимальных решений? Прежде всего, ясно, что для задач непрерывной оптимизации сама постановка вопроса неправильна, – оптимальное решение непрерывной задачи содержит обычно целую окрестность решений со сколь угодно близкими значениями целевой функции. Но, кроме этого очевидного аргумента, нужно отметить, что задачи дискретной оптимизации более «капризны», трудоемкость их решения резко меняется при малейшем изменении задачи. Практические задачи трудны очень часто, поэтому при их решении некоторыми существенными ограничениями могут пренебречь, а упрощение математической модели приведет к тому, что полученное оптимальное решение окажется нереализуемым из-за *внемодельных факторов*, то есть тех условий, которые не стали включать в математическую модель. Кажется естественным такой подход: перебирать допустимые решения с постепенным ухудшением значения целевой функции до тех пор, пока не встретится решение, приемлемое с точки зрения истинных ограничений.

Задачи перебора субоптимальных решений встречались в научной литературе многократно. Как правило, в этих работах вопрос исследовался для конкретных оптимизационных задач, причем для очень узкого набора.

Наш подход будет более общим. Предлагаются средства, пригодные для широкого класса задач. Этот подход заключается в выделении набора операций, с помощью которых процесс перебора решений нужной задачи компонуется из более простых процессов. Элементы такого подхода уже встречались, например, у Е. Лаулера [1], который отметил важность операции слияния, и у Миниеки и Шира [2], где операции слияния и суммирования списков введены для операций над матрицами списков в связи с попыткой переноса известного метода Флойда на задачу построения матрицы списков *k*-кратчайших путей. Некоторые другие интересные ссылки мы приведем в конце статьи.

Развитие общего подхода предусматривает выяснение возможных различий в постановке задачи. Например, вычислительная сторона задачи существенно меняется в зависимости от того, требуется перебор всех допустимых решений задачи или только некоторого их количества, – в последнем слу-

чае могут быть приняты специальные меры, повышающие эффективность.

Отметим еще, что интерес к перебору субоптимальных решений должен повышаться и действительно повышается в связи с крупными изменениями используемых вычислительных средств. С одной стороны, это резкое улучшение характеристик вычислительной техники, ее быстродействия и оперативной памяти, появления многопроцессорных систем. С другой стороны, появление новых возможностей организации вычислительных процессов, заложенных в более эффективных языках программирования.

Мы начнем изложение с напоминания (на простых примерах) принципов динамического программирования и с перебора субоптимальных решений в нем. Это даст нам фундамент для введения некоторых базовых понятий, а затем для основного аппарата перебора – перечисляющих процессов (назовем их для удобства *перечислителями*). и операций над ними. Работа с этим аппаратом демонстрируется затем на нескольких конкретных задачах разной сложности.

После этого мы будем готовы к рассмотрению вопросов программной реализации перечислителей в духе объектно-ориентированного программирования.

Эта статья во многом следует моей книге [3].

ПЕРВОНАЧАЛЬНЫЕ ПРИМЕРЫ

Перебор субоптимальных решений в задачах дискретной оптимизации начался с динамического программирования. Еще в работе создателя динамического программирования Ричарда Беллмана и его соавтора Р. Калабы [4], в 1960 г. рассматривалась задача перебора путей в графе (см. также [5]). С этой задачи и с подхода Беллмана-Калабы мы и начнем. Рассмотрим традиционный подход динамического программирования.

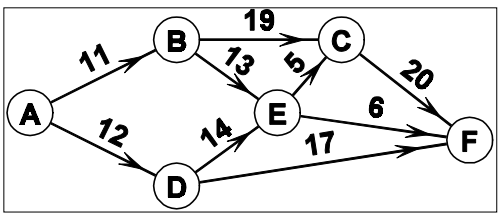

Рис. 1. Граф для примера

1. Кратчайший путь в графе без контуров

Эту задачу можно считать хорошо известной, поэтому ее удобно использовать для введения в методы рекуррентного решения оптимизационных задач. Итак, пусть задан ориентированный граф без контуров $\langle M, N\rangle$ с множеством вершин M и множеством дуг N. Заданы длины дуг, и длина пути определяется как сумма длин входящих в путь дуг. Требуется найти кратчайший путь с заданными началом i_0 и концом i_e.

Пример. Рассмотрим простой пример. Задан граф без контуров с шестью вершинами, каждой дуге которого сопоставлена положительная длина (рис. 1). Требуется найти кратчайший путь от вершины A до вершины F.

Прежде всего (и это важная черта динамического программирования) расширим задачу и будем искать кратчайшие пути от A до всех вершин графа. Обозначим длину пути до вершины i, $i \in \{A, B, C, D, E, F\}$ через $v(i)$. Естественно принять $v(A) = 0$. Все множество путей, проходящих из A в F, можно разбить на три подмножества по тому признаку, какая дуга будет в пути последней, ведущая из C, из D или из E. К примеру, путь $ADEF$ попадет в третью группу, так как он кончается дугой EF. Можно искать путь минимальной длины в каждом из этих множеств отдельно, а затем выбрать минимум из минимумов (по латыни это *minimum minimorum*). В естественных обозначениях

$$v(F) = \min\{v_C(F), v_D(F), v_E(F)\}.$$

Но очевидно,

$$v_C(F) = 20 + v(C),\ v_D(F) = 17 + v(D),\ v_E(F) = 6 + v(E),$$

то есть

$$v(F) = \min\{20 + v(C), 17 + v(D), 6 + v(E)\}.$$

Аналогичное соотношение можно написать для любой вершины, кроме начальной,

$$v(i) = \min\{c(j) + v(\mathbf{beg}\ j) \mid \mathbf{end}\ j = i\}. \quad (1)$$

Здесь **beg** *j* и **end** *j* – соответственно начало и конец дуги *j*, а $c(j)$ – ее длина.

Применение уравнения (1) прямо приводит к требуемому результату. Имеем:

$$v(B) = 11 + v(A) = 11,\ v(D) = 12,$$

так как $v(A) = 0$,

$$v(E) = \min\{13 + v(B), 14 + v(D)\} = 24,$$

$$v(C) = \min\{19 + v(B), 5 + v(E)\} = 29,$$

откуда окончательно $v(F) = 29$.

Уравнение (1) называется *уравнением динамического программирования* или *уравнением Беллмана*, а функция v – функцией Беллмана.

Задача о кратчайшем пути может решаться с помощью разных подходов, и впоследствии мы будем говорить об очень эффективном алгоритме Дейкстры для ее решения на произвольном графе, но здесь мы говорим об этой задаче в связи с содержащимся в ней *процессом динамического программирования*, то есть специфическим процессом последовательного принятия решений. Множество вершин графа M – это множество *состояний* процесса, A – начальное состояние. Дуги, выходящие из вершины $i \in M$, – это *решения*, которые можно принимать в состоянии i. Принятие каждого решения j влечет затраты $c(j)$ и переводит процесс в состояние $i' =$ **end** j.

Потеря общности в сравнении с общей схемой динамического программирования лишь в специальном виде целевой функции, по которой оценивается траектория процесса, – здесь она равна сумме затрат от принятых решений. Но для наших целей этого частного случая вполне достаточно. Более того, чтобы увидеть некоторые особенности экономии вычислений в задачах динамического программирования, мы рассмотрим еще более простой случай.

2. Наилучший линейный раскрой

Рассмотрим теперь известную задачу о выборе наилучшего раскроя линейного сырья заданной длины l_0 на детали меньшей длины $l[i]$, $i \in 1:m$, для получения наибольшего дохода (цены деталей p_i известны) [6].[1]

Пусть раскрою подлежит некоторое сырье, имеющее одно линейное измерение; например, пусть рулон бумаги, вышедший с бумагоделательной машины, необходимо разрезать на рулоны меньшего размера так, чтобы доход от их продажи был максимальным.

В математических терминах, требуется найти неотрицательные и целочисленные $x[i]$, $i \in 1:m$, – количества рулонов каждого i-го вида, удовлетворяющие условию:

$$\sum_{i \in 1:m} l[i] \times x[i] \le l_0$$

и максимизирующие

$$\sum_{i \in 1:m} p[i] \times x[i].$$

Обозначим искомый максимум через $v(l_0)$. Если рассматривать его как функцию от размера сырья, можно связать значения этой функции для разных размеров уравнением

$$v(\lambda) = \max\{c_i + v(\lambda - l[i]) \mid i \in 1:m,\ l[i] \le \lambda\},$$

где λ – длина сырья (переменная, при этом нас интересует $v(l_0, m)$), $v(l)$ принимается равным 0, если нет деталей длины меньшей λ.

Фактически мы имеем дело с графом $\langle M, N\rangle$, где $M = 0:l_0$, то есть вершинами являются все возможные промежуточные размеры сырья, а дуги соответствуют возможным *резам*: когда от рулона размера λ отрезается рулон размера $l[i]$, получаем рулон размера $\lambda - l[i]$, – такая возможность определяет дугу, идущую из состояния $\lambda - l[i]$ в состояние λ.

Это тоже *уравнение Беллмана*. Его можно использовать непосредственно для вычисления значений $v(\lambda)$, начиная от малых значений аргумента и постепенно их увеличивая. Однако эта исходная вычислительная схема может быть существенно улучшена. Рассмотрим разные формы уравнения Беллмана

Первое из усовершенствований связано с тем, что можно сразу определять, сколько раз должна быть отрезана последняя, m-ая деталь, а затем раскраивать остаток сырья

[1] Первое издание этой книги вышло в 1951 г., когда термина «динамическое программирование» еще не существовало. Авторы предугадали динамическое программирование.

на оставшиеся детали. Обозначив через $v(\lambda, k)$ максимальный доход от раскроя сырья длины λ на детали первых k размеров (нас интересует $v(\lambda, k)$), получаем следующее уравнение Беллмана

$$v(\lambda, k) = \max \{c_k s + v(\lambda - l[k]\, s, k-1) \mid 0 \le l[k]\, s \le 1\}.$$

Уравнение можно еще упростить, сделав выбор количества резов последней детали последовательным принятием решения

$$v(\lambda, k) = \max \{v(\lambda, k-1), c_k s + v(\lambda - l[k], k)\}.$$

Здесь нужно выбирать между отказом от дальнейшего использования k-й детали и однократным ее выбором с последующим решением о дальнейших выборах. Легко видеть, что этот вариант уравнений в вычислительном отношении существенно легче предыдущего.

Вычисление функций $v(\lambda, k)$ может быть существенно упрощено, если воспользоваться тем, что при любом k функция $v(\lambda, k)$ – кусочно-постоянная неубывающая функция от λ и, значит, может задаваться списком скачков. Вычислительный процесс может быть описан как процесс переработки списка скачков k–1-й итерации в список k -й итерации [7].

3. «Ленивое исполнение»

Однако и исходный вариант уравнения Беллмана не безнадежен в вычислительном отношении. Можно организовать *каскадное* вычисление значений функции Беллмана, вычисляя ее только в тех точках, которые были запрошены непосредственно или в ходе вычисления в других точках. Такой способ вычислений называется по-английски *lazy evaluation – ленивое исполнение*.

Принцип ленивого исполнения: не вычислять ничего до тех пор, пока это не станет необходимо. В этом методе вычисление идет только для потребовавшихся значений и только тогда, когда нужно. Этим способом организации вычислений мы будем активно пользоваться для поиска субоптимальных решений.

4. Оптимизационная свертка

Уравнение Беллмана можно записывать более компактно, вводя специальные операции, например операцию оптимизационной свертки. Термин *свертка* (convolution) заимствован из математического анализа, где сверткой двух последовательностей $A = \{a_k\}$ и $B = \{b_k\}$ называется последовательность

$$C = \{c_k\}, \text{ где } c_k = \sum\nolimits_{i \in 0:k} a_i \cdot b_{k-i}.$$

Оптимизационная свертка может быть *мультипликативной*, больше похожей на классическую, и *аддитивной*, более естественной, которую мы и выберем (ср. по этому поводу [8] и [9]). Итак, оптимизационной сверткой двух последовательностей $A = \{a_k\}$ и $B = \{b_k\}$ называется последовательность $C = \{c_k\}$, где

$$c_k = \max \{a_i + b_{k-i} \mid i \in 1:k\}.$$

Это определение симметрично и может быть очевидным образом распространено на любое число слагаемых. Обозначим такую свертку знаком $\oplus$, так что $C = A \oplus B$.

Пусть теперь задано t последовательностей $A_1, A_2, \ldots, A_t$ и требуется найти их свертку $A_1 \oplus A_2 \oplus \ldots \oplus A_t$. Введем последовательность последовательностей V_s: $V_1 = A_1$, $V_s = V_{s-1} \oplus A_s$, $s > 1$. Последовательности V_s – это фактически функции Беллмана, определяемые при $s > 1$ уравнением

$$v_s[k] = \min \{a_s[i] + v_{s-1}[k-i] \mid i \in 0:k\}.$$

В следующем параграфе для операции свертки найдется применение.

5. Схема Бертеле - Бриоши: выбор поддерева в дереве

Рассмотрим еще одну разновидность динамического программирования, которую ее авторы, итальянские математики Бертеле и Бриоши, назвали *Nonserial dynamic programming* (то есть *не последовательное*, в противоположность обычному *последовательному,* в котором есть параметр, аналогичный времени) ([10], см. также [7]).

Здесь мы ограничимся примером, который важен для дальнейшего изложения и интересен тем, что, несмотря на всю свою простоту, не укладывается в схему, использованную раньше.

Пусть задано ориентированное дерево $\langle M, N \rangle$ с корнем $r \in M$. Это значит, что в графе число дуг на единицу меньше числа вер-

шин, $|N| = |M| - 1$, и имеется путь из корня в любую другую вершину. Пусть дугам графа сопоставлены положительные веса w_j, $j \in N$. Требуется по заданному положительному числу $m' < |M|$ найти ориентированное дерево $\langle M', N' \rangle$ с корнем в r и такое, что $|M'| = m'$, $N' \subset N$, и сумма весов дуг из N' минимальна.

Пусть $d = |M| - m'$. Наша экстремальная задача, очевидно, может быть заменена задачей об удалении из дерева d дуг таким образом, чтобы оставалось корневое дерево с корнем в r и была максимальна сумма весов удаленных дуг.

Отметим, что каждой вершине i в графе соответствует свое поддерево Γ_i, в котором эта вершина является корнем. Будем рассматривать задачу о выборе поддерева с удалением d дуг минимального суммарного веса для всех деревьев Γ_i и произвольных d, используя принципы динамического программирования. Пусть $v(i, d)$ – искомый минимальный вес в этой задаче, а V_i – последовательность таких минимумов для всех возможных d, от 0 до числа дуг в дереве Γ_i. Пусть $\overline{V}_i$ – это последовательность V_i, к которой добавлен еще один элемент – последний элемент, увеличенный на вес дуги i, входящей в вершину i (мы как бы рассматриваем дерево Γ_i, наращенное на дугу i, и для этого дерева решаем такую же задачу).

Теперь осталось отметить, что, во-первых, $\overline{V}_i = \{0\}$ для конечных вершин дерева (листьев), во-вторых, нас интересует $\overline{V}_r$ (точнее, только $v(r, d)$), и, наконец, в-третьих, что

$$\overline{V}_i = \oplus_{j:\, \mathrm{beg}\, j = i} \overline{V}_j .$$

Это тоже уравнение Беллмана, но в обобщении Бертеле-Бриоши. Использование операторной записи делает его очень компактным. Выполним небольшое упражнение, чтобы увидеть функцию Беллмана более наглядно.

Пример. Рассмотрим дерево с корнем R, изображенное на рис. 2. Вершины B, D, G, H, J, K, L – листья, для каждой из них последовательность V состоит из одного элемента 0. Для вершины C последовательность состоит уже из двух элементов $V_C = \{0, 4\}$, для вершины F из трех $V_F = \{0, 8, 11\}$, для I из четырех $V_I = \{0, 9, 14, 16\}$.

Теперь перейдем к более сложным расчетам. Для вычисления V_A нужно продолжить последовательность V_C до $\overline{V}_{AC} = \{0, 4, 11\}$, а уже затем найти $V_A = \overline{V}_{AB} \oplus \overline{V}_{AC}$. Имеем $v_A[0] = 0$, $v_A[1] = \max\{4, 15\} = 15$, $v_A[2] = \max\{11, 15 + 4\} = 19$, $v_A[3] = 15 + 11 = 26$.

Вычисление V_E более громоздко. При каждом k нужно вычислить все суммы вида $v_{EF}(i) + v_{EI}(k - i)$, и минимальную из них принять в качестве $v_E(k)$. Имеем

$v_E[0] = 0$,
$v_E[1] = \max\{8, 9\} = 9$,
$v_E[2] = \max\{0 + 14, 8 + 9, 11 + 0\} = 17$,
$v_E[3] = \max\{0 + 16, 8 + 14, 11 + 9, 18 + 0\} = 22$,
$v_E[4] = \max\{0 + 20, 8 + 16, 11 + 14, 18 + 9\} = 27$,
$v_E[5] = \max\{8 + 20, 11 + 16, 18 + 14\} = 32$,
$v_E[6] = \max\{11 + 20, 18 + 16\} = 34$,
$v_E[7] = 18 + 20 = 38$.

Далее нужно вычислить промежуточные функции $\overline{V}_{RA}$ и $\overline{V}_{RE}$, а затем по ним окончательно построить v_R.

k-ОПТИМАЛЬНЫЕ РЕШЕНИЯ В ДИНАМИЧЕСКОМ ПРОГРАММИРОВАНИИ

1. Операция слияния упорядоченных массивов

В дальнейшем часто будет использоваться *операция слияния* (merge). Эта операция была хорошо известна во времена последовательных файлов на магнитных лентах, а

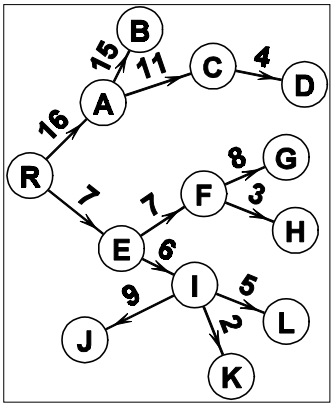

Рис. 2. Дерево с корнем

Табл. 1

Шаг	1	2	3	4	5	6	7	8	9	10	11
A	5	9	17	26	31						
B	2	6	8	12	27	29					
C	2	5	6	8	9	12	17	26	27	29	31
k_1	1	1	2	2	2	3	3	4	5	5	5
k_2	1	2	2	3	4	4	5	5	5	6	

сейчас остается известной благодаря, прежде всего, сортировке Неймана.

Пусть заданы две монотонно неубывающие последовательности $A = \{a_i\}$ и $B = \{b_i\}$ и требуется соединить их в одну последовательность $C = \{c_i\}$. Это соединение выполняется параллельным «чтением» обеих последовательностей от начала: на каждом шаге из прочтенных элементов выбирается наименьший, для того чтобы записать его в последовательность C, а вместо взятого элемента читается новый из той же последовательности. Так выглядит слияние файлов последовательного доступа.

В случае, если последовательности $A = \{a_i\}$ и $B = \{b_i\}$ записаны в массивы, для организации слияния нужно помнить на каждом шаге индексы текущих элементов читаемых массивов и обновлять их. Например, если $A = \{5, 9, 17, 26, 31\}$ и $B = \{2, 6, 8, 12, 27, 29\}$, изменение индексов в ходе слияния выглядит так, как показано в табл. 1.

Индексы k_1 и k_2 показывают здесь степень использованности (*сработанности*) массивов A и B на каждом шаге процесса слияния.

Очевидно, что аналогично можно организовать слияние нескольких последовательностей, причем можно организовывать слияние по-разному, например, слить последовательности в небольших группах, а затем слить вместе получившиеся результаты.

Табл. 2

Шаг	1	2	3	4	5	6
k_C	1	1	1	1	2	2
k_D	1	2	2	2	2	2
k_E	1	1	2	3	4	4

2. Перебор k-кратчайших путей в графе без контуров

Обратимся снова к задаче поиска кратчайшего пути в графе без контуров и попробуем в ней искать пути, следующие после кратчайшего. Используем уже рассмотренный пример.

Найденный кратчайший путь ADF проходит через вершину D. Очевидно, что второй кратчайший путь до F – это либо кратчайший путь через C, либо кратчайший путь через E, либо второй кратчайший путь через D. В функции v добавим еще один аргумент, номер пути. Тогда[1]

$$v(F, 2) = \min \{ v_C(F, 1), v_D(F, 2), v_E(F, 1)\},$$

или

$$v(F, 2) = \min \{20 + v(C, 1), 17 + v(D, 2), 6 + v(E, 1)\}.$$

Аналогично

$$v(D, 2) = \min \{12 + v(A, 2)\} = \infty,$$

так как $v(A, 2) = \infty$. Поскольку $v(C, 1) = 29$, $v(E, 1) = 24$, то $v(F, 2) = 30$, эта длина достигается на пути $ABEF$.

Третий кратчайший путь находится из соотношения

$$v(F, 3) = \min \{20 + v(C, 1), 17 + v(D, 2), 6 + v(E, 2)\},$$

он имеет длину $v(F, 3) = 32$, сам путь $ADEF$.

Для остальных путей запишем только их длины и соотношения, из которых они получены:

$$v(F, 4) = \min \{20 + v(C, 1), 17 + v(D, 2), 6 + v(E, 3)\} = 49,$$

$$v(F, 5) = \min \{20 + v(C, 2), 17 + v(D, 2), 6 + v(E, 3)\} = 50,$$

$$v(F, 6) = \min \{20 + v(C, 3), 17 + v(D, 2), 6 + v(E, 3)\} = 51.$$

Выпишем из соотношений для $v(F, \cdot)$ индексы k_i в функциях $v(i, \cdot)$ в виде таблицы (см. табл. 2).

Получается очень похоже на таблицу индексов сработанности при слиянии двух мас-

[1] Такие соотношения приводились еще в 1960 г. в уже упоминавшейся работе Р. Беллмана и Р. Калабы [4].

сивов. На самом деле это и есть слияние, слияние массивов значений в предшествующих вершинах графа, к которым «прибавлены» длины дуг.[1] В каждой из этих вершин, кроме начальной, ее массив можно также записать как слияние других массивов. Значит, все значения можно получить так просто (это так похоже на программу, что используем «программистский» шрифт):

```
aA = [0]
aB = 11 + aA
aD = 12 + aA
aE = merge { (13+aB), (14+aD) }
aC = merge { ( 5+aE), (19+aB) }
aF = merge { ( 6+aE), (17+aD), (20+aC) }
```

Очень просто! Можно взять любой граф без контуров, написать аналогичные соотношения для массивов значений во всех вершинах и получить практически готовую программу. Более того, существуют такие языки программирования, для которых программа – это почти такая запись.

Можно даже не рисовать граф, а ограничиться перечислением его дуг, лучше прямо в виде входящих звезд по каждой вершине.

ПЕРЕЧИСЛИТЕЛИ И ОПЕРАЦИИ НАД НИМИ

Наконец-то, после всех вводных рассмотрений перейдем к главному предмету. Мы введем новый тип объектов, логических и программистских, и будем рассматривать наши алгоритмы в терминах операций над этими объектами.

Этот объект называется *перечислителем*. Перечислитель может *вызываться*, и при каждом вызове он сообщает некоторое числовое *значение* и сопровождающую это значение текстовую информацию, называемую *управлением* или *решением*. Числовые значения, возвращаемые при последовательных вызовах, расположены в порядке неубывания. По исчерпании значений перечислитель может сообщить, что его работа закончилась.

Так описывается самая простая разновидность перечислителя – последовательный одноразово используемый процесс.

Для тех случаев, когда одно и то же значение может потребоваться несколько раз, целесообразно предусмотреть *многоразово используемый процесс*, вызов которого происходит с указанием номера требуемого значения. Такой перечислитель может рассматриваться как надстройка над простым. Эта надстройка ведет базу данных с вычисленными значениями подчиненного перечислителя, выдает из этой базы внешнему миру запрашиваемые значения и при необходимости сама вызывает подчиненный перечислитель для пополнения своей базы данных до требуемого уровня.[2]

Третий вариант – регистрируемый многоразово используемый процесс, который получает внешнее имя, и по этому имени может быть найден другими перечислителями.[3]

Рассмотрим операции, позволяющие создавать новые перечислители, в том числе из уже имеющихся.

1. Перечислитель – массив

Самый простой перечислитель может быть задан массивом – монотонной последовательностью значений (неубывающей, если исходная экстремальная задача на максимум, и невозрастающей, если на минимум). Этот массив хранится во *внутренней информации* перечислителя вместе с *индексом очередного значения*. На каждый вызов процесс отвечает значением, соответствующим этому индексу, и увеличивает индекс на единицу. При выходе индекса за границу массива перечислитель сообщает об отсутствии новых значений.

---

[1] На роль операции слияния в процессах перебора субоптимальных решений впервые обратил внимание Е. Лаулер [1].

[2] Нужно ясно представлять себе, что при вызове такого перечислителя номер требуемого значения определяется извне. Однажды я встретился с неправильным пониманием этой конструкции как возврата значения вместе с его номером.

[3] Первоначально я рассматривал только два типа процессов. Предлагаемая здесь классификация появилась по предложению аспиранта М.Д. Папоркова.

Удобно иметь какое-нибудь обозначение для такого процесса. Пусть это будет ARRAY(a_1,…,a_k).

2. Преобразование значений перечислителя

Пусть заданы некоторый перечислитель P и монотонно неубывающая функция f, преобразующая его значения в другие значения. Описываемая операция заключается в том, что процесс P вызывается очередной раз и полученное им значение подставляется в функцию f. Результатом операции является перечислитель, возвращающий при каждом вызове преобразованные ответы P.

Наиболее употребительное монотонное преобразование – прибавление некоторой константы c. Мы таким преобразованием и ограничимся, назовем его *сдвигом* и будем записывать так: Q = PLUS(P, c).

3. Слияние перечислителей

Разумеется, дальше должно пойти *слияние* процессов, о котором уже много говорилось.

Пусть заданы два процесса P и Q. Рассмотрим процесс, который вызывает оба процесса, выбирает из полученных значений наименьшее и возвращает в ответ на свой вызов. Разумеется, при каждом своем следующем вызове процесс вызывает только тот из подчиненных перечислителей, значение которого было использовано, и не вызывает перечислителей, завершивших работу.

Обозначим операцию слияния двух перечислителей так: MERGE(P, Q). Это же обозначение мы сохраним и для операции слияния нескольких перечислителей: MERGE(P_1, …, P_k).

4. Задержанное слияние

Следующая операция – это небольшая модификация слияния. Предположим, что, сливая два процесса, P и Q, мы знаем, что даже первое значение, вырабатываемое процессом Q, будет не меньше некоторой константы c. В этом случае, до тех пор пока значения процесса P не превосходят этого c, процесс Q можно даже и не вызывать и, значит, даже не инициализировать. Таким образом, здесь слияние происходит с первоначальной *задержкой*.

Обозначим операцию задержанного слияния двух перечислителей так: DAMERGE(P, Q, c). Здесь буквы D и A от Delayed Absolute, потому что задержка определяется абсолютной величиной значения.

Есть еще один вариант задержанного слияния, при котором значение задержки относительно, оно отсчитывается не от нуля, а от первого значения процесса P. Эта операция будет обозначаться DMERGE(P, Q, c).

Обе эти операции появились уже в результате развития нашего подхода к алгоритмам перебора как к процессам. Они существенно повысили эффективность разрабатываемых алгоритмов.

5. Фильтрация значений перечислителя

Мне казалось, и продолжает казаться, что должна быть полезна операция фильтрации, в которой элементы, получаемые на входе операции, проходят дополнительную проверку и в некоторых случаях отвергаются, так что чтение новых элементов идет до получения «правильного» элемента.

6. Сложение перечислителей

Пусть заданы два процесса P и Q. Представим себе, что мы располагаем всеми значениями обоих процессов. Рассмотрим всевозможные суммы этих значений – по одному из каждого набора – и введем процесс, который вырабатывает эти суммы в порядке возрастания (неубывания). Такой процесс будет называться *суммой* перечислителей P и Q и обозначаться SUM(P, Q).

Пример. Если P = [4, 14, 29, 43], Q = [11, 16, 25], то SUM(P, Q) = [15, 20, 25, 29, 30, 39, 40, 45, 54, 54, 59, 68]

Можно рассматривать и сумму нескольких перечислителей, распространяя на нее то же обозначение: SUM(P_1, …, P_k).

Операция суммы нужна для перебора значений функции, заданной на прямом произведении двух (или нескольких) множеств и равной сумме функций от сомножителей. Например, если каждый процесс P_i – это

массив из двух значений P_i = ARRAY (0, a_i), то SUM (P_1, …, P_k) перебирает все суммы набора $\{a_1, \ldots, a_k\}$ по всем его подмножествам.

Конечно, не следует слишком соблазняться легкостью, с которой пишутся подобные соотношения. Мы получаем здесь множество, в котором 2^k элементов, и это число остается большим, как его ни записывай. Но подобные операции вполне пригодны для получения некоторой части множества.

Осталось посмотреть, как можно «лениво» реализовать операцию суммирования процессов, не выписывая заранее всех значений обоих процессов. Некоторые надежды приносит тот факт, что для получения первого значения суммы процессов нужны только первые их собственные значения.

А для второго значения суммы нужны еще и вторые значения слагаемых. Если обозначить i-ые значения процессов соответственно через p_i и q_i, то мы должны выбирать второе значение суммы как максимум из $p_1 + q_2$ и $p_2 + q_1$. Заметим, что сумма $p_2 + q_2$ в этом рассмотрении не участвует, она *доминируется* двумя другими суммами. Вопрос о недоминируемых парах индексов рассмотрим отдельно.

7. Граница Парето

Пусть заданы два отрезка натурального ряда чисел $K = 1:k$ и $L = 1:l$, и R – прямое произведение этих множеств, $R = K \times L = \{(i, j) \mid i \in 1:k, j \in 1:l\}$. Возьмем какое-либо подмножество $S \subset R$.

Скажем, что пара $(i, j) \in R$ *доминирует* пару $(i', j') \in R$, если $i \le i'$ и $j \le i'$.

Границей Парето множества S называется такое его подмножество B, что любой элемент $(i, j) \in S \setminus B$ доминируется каким-нибудь элементом из B и никакой элемент из B не доминируется другим[1] .

Теперь можно вернуться к процессу суммирования. При каждом вызове i имеется множество еще не использованных пар индексов $S_i \setminus R$, выбирается один из них, скажем, s_i, и следующее множество получается удалением выбранного индекса, то есть $S_{i+1} = S_i \setminus \{s_i\}$. Легко можно доказать, что индекс s_i следует выбирать из i-й границы Парето B_i. Действительно, если пара (i, j) доминирует (i', j'), то $p_i + q_j \le p_{i'} + q_{j'}$.

Программная поддержка действий с границей Парето оказалась очень простой, но говорить о ней сейчас мы не будем. Посмотрим только, как меняется граница Парето в приведенном выше примере.

Продолжение примера. Для наглядности будем определять координаты не индексами, а значениями. Первоначально имеем в границе Парето одну пару B_1 = (4, 11). Выбор этой пары с суммой 15 изменяет границу $B_2 = \{(14, 11), (4, 16)\}$. Пара с суммой 20 выбирается, и мы получаем $B_3 = \{(14, 11), (4, 25)\}$. Затем $B_4 = \{(29, 11), (14, 16), (4, 25)\}$ и т. д.

ИСПОЛЬЗОВАНИЕ

Я использовал этот подход в ряде классических экстремальных задач (при замене их задачами перебора) и в некоторых прикладных задачах. Введенные в статье операции легко программируются в любом языке, допускающем объектно-ориентированный стиль программирования. Очень удобно было использовать для экспериментального программирования язык функционального программирования Miranda ([11], см. пример использования в [12]), интерпретатор для которого мне любезно предоставил Д. Тернер.

Надеюсь, что смогу написать об этом в продолжении статьи.

[1] Вилфред Парето (1848–1923) – итальянский экономист и социолог, который ввел понятие точки, оптимальной по нескольким показателям (распределение благ между несколькими лицами), как точки, у которой нельзя улучшить один показатель, не ухудшив других. Наша граница – это множество точек, оптимальных по Парето. В сталинские времена имя Парето было под запретом: утверждалось, что он – «любимец Муссолини», хотя Муссолини (1883–1945) только слушал лекции Парето (1902), еще не будучи душе – фашистским вождем, а премьер-министром стал в 1922 г. Муссолини был человек умный и, действительно, Парето ценил, но это не переносит на Парето ответственность за грехи итальянских фашистов.

Литература

1. *Lawler E.L.* A procedure for computing the K-best solutions to discrete optimization problems and its application to the shortest path problem // Management Science, 1972. № 18. P. 401–405.

2. *Minieka E., Shier D.* A note on an algebra for the k-best routes in a network // J. Inst. Math. Appl., 1973. № 11. P. 145–149.

3. *Романовский И.В.* Субоптимальные решения, Петрозаводск: Изд-во Петрозав. ун-та, 1998.

4. *Bellman R., Kalaba R.* On k-th best policies // J. of SIAM, 1960. № 8. P. 582–588.

5. *Беллман Р., Дрейфус С.* Прикладные задачи динамического программирования. М.: Наука, 1965.

6. *Канторович Л.В., Залгаллер В.А.* Рациональный раскрой промышленных материалов. Изд. 3-е. СПб.: Невский диалект, 2012.

7. *Романовский И.В.* Алгоритмы решения экстремальных задач. М.: Наука, 1977.

8. *Bellman R., Karush W.* On a new functional transformation // Bull. Amer. math. soc., 1961. № 67. 5. P. 501–503.

9. Р*омановский И.В.* Некоторые замечания о функциональном преобразовании Беллмана–Каруша // Вестник СПб. университета, сер. мат., 1962. Вып. 3. С. 148–150.

10. *Bertelo U., Brioshi F.* Non-Serial Dynamic Programming. N.Y.: Academic Press, 1972.

11. *Turner D.A.* An overview of Miranda. Research Topics in Functional Programming, D.A. Turner, ed., Addison-Wesley, 1990. P. 1–16.

12. *Романовский И.В., Стукалов Д.Ю.* Алгоритмы перебора неизоморфных деревьев // Вестник СПб. университета, сер. 1, 1997. Вып. 1. С. 46–52.

Abstract

The paper presents an approach to enumeration of suboptimal solutions, based on introduction of special processes of enumeration – «enumerators», which solve problems of that kind for various smaller problems. Some special operations presented in the paper allow us to make the required enumerators of simpler ones. Some examples are presented.

Keywords: suboptimal solutions, discrete optimization, dynamic programming.

Романовский Иосиф Владимирович, доктор физико-математических наук, профессор, заведующий кафедрой исследования операций математико-механического факультета СПбГУ, josephromanovsky@gmail.com