УДК 518.5

ПРЕПРОЦЕССИНГ В МОДУЛЯРНЫХ АЛГОРИТМАХ

© В.Н. Казаков

Kazakov V.N. Preprocessing in modular algorithms. The method is proposed of computation organization in computational problems where the Chinese theorem of remainders is applied. At creation of computational arrays connected to lists of simple modules, possible passes of the chosen modules are taken into account. It is exemplified by a resultant computation by Brown's method.

Рассмотрим задачу решения определенной системы линейных уравнений с целыми коэффициентами $Ax = b$, где $A = (a_{ij})$ матрица порядка n. Вектор решения такой системы образован дробями, у которых числители и знаменатели – это целые числа.

Существует много прямых способов решения систем, каждый из них приводит к росту знаменателей и числителей дробей, участвующих в промежуточных вычислениях. Поэтому, если сложность метода оценивается, например, как $O(n^3)$ в количестве операций над элементами матрицы, то в количестве машинных слов она оценивается как $O(n^5)$.

Для уменьшения сложности вычислений можно использовать модулярный метод. Суть этого метода состоит в том, что выбирается несколько простых модулей, по каждому модулю решается система, и по этим решениям находится решение системы в рациональных числах. Однако если случайно выбранный модуль оказался делителем определителя основной матрицы системы, то найти решение системы по такому модулю невозможно. Поэтому, если имеется список заранее выбранных модулей, то в процессе вычислений приходится некоторые из этих модулей пропускать. Список простых модулей выбирается так, чтобы их произведение превышало удвоенную оценку определителя матрицы, полученной с помощью неравенства Адамара. Неравенство Адамара для квадрата модуля детерминанта матрицы

$$\alpha = \prod_{i=0}^{n}\left(\sum_{j=0}^{n}\left(a_{ij}\right)^2\right).$$

Рассмотрим схему Ньютона восстановления числа по остаткам.

Обозначим p_i , $i = 1, 2, \ldots N$ – простые числа, r_i – остатки от деления на p_i искомого числа a, $i = 1, 2, \ldots N$.

Пусть

$$q_i = (p_1 p_2 \cdots p_i)^{-1} \bmod p_{i+1},$$

$$s_1 = r_1,\ s_2 = r_1 + (r_2 - \tilde{S}_1)\, q_1 p_1,\ \ldots,$$

$$s_i = r_1 + (r_2 - \tilde{S}_1)\, q_1 p_1 + \ldots + (r_i - \tilde{S}_{i+1})\, q_{i-1} p_1 p_2 \cdots p_{i-1}$$
$$(2 < i \le N),$$

$$\tilde{S}_i = s_i \bmod p_{i-1}\ (1 \le i \le N-1)$$

Тогда $a = s_N \bmod p_1 \cdots p_N$.

Главным недостатком данного алгоритма является то, что при вычислении q_i приходится прибегать к вычислениям над большими целыми числами.

Здесь предлагается метод организации модулярных вычислений, включающий оптимальный препроцессинг, учитывающий возможный пропуск модулей в заранее выбранном списке в процессе вычислений.

Данные хранятся в четырех целочисленных массивах:

1. prims – простые модули, содержащие не более 28 бит: $p_1, p_2, \ldots, p_i$;
2. mulPrims – их произведения в формате BigIntrger: $p_1, p_1 * p_2, \ldots, p_1 * p_2 * \ldots * p_i$;
3. inversMulPrims – произведения чисел обратных к произведениям первых n простых чисел по $n+1$-му простому модулю (28 бит): $(p_1)^{-1} \bmod p_2$, $(p_1 * p_2)^{-1} \bmod p_3, \ldots, (p_1 * p_2 * \ldots * p_i)^{-1} \bmod p_{i+1}$;
4. mulInversPrims – произведения чисел обратных к произведениям первых n простых чисел по $n+1$-му простому модулю в формате BigInteger: $(p_1)^{-1} \bmod p_2$, $((p_1)^{-1} \bmod p_3) * ((p_2)^{-1} \bmod p_3), \ldots,$ $((p_1)^{-1} \bmod p_i) * ((p_2)^{-1} \bmod p_i) * \ldots * ((p_i)^{-1} \bmod p_i), \ldots$

Массивы создаются фиксированного размера, например, 64. Если простых модулей не хватает, массивы увеличиваются каждый раз на следующие 32 элемента.

Алгоритм восстановления числа по его остаткам от деления на простые модули теперь будет иметь следующий вид.

1. Если ни один модуль из выбранных не исключается, то вычисления производятся согласно вышеуказанной схеме, при этом q_i и произведения простых модулей $p_1 * p_2 * \ldots * p_i$ берутся из массивов inversMulPrims и mulPrims;
2. Пусть $\{l_1, l_2, \ldots, l_m\}$ – номера пропущенных модулей из массива prims. Алгоритм восстановления до участия первого пропущенного модуля l_1 производится также как и в первом пункте.

Для дальнейших вычислений нельзя использовать готовые q_i и произведения простых модулей $p_1 * p_2 * \ldots * p_i$, т. к. они содержат в себе исключенные модули. Поэтому в массиве mulInversPrims выбирается элемент

mulInversPrims $(i) = ((p_1)^{-1} \bmod p_i) * ((p_2)^{-1} \bmod p_i) * \ldots * ((p_{i-1})^{-1} \bmod p_i)$,

где i – первый из модулей после l_1-го, который не пропущен. Тогда при $j = l_1$ получим искомый элемент q

q = mulInversPrims (i)/($(p_j)^{-1}$ mod p_i) mod p_i.

Тем самым всего за две операции получается следующее значение q. Аналогично получаются и следующие элементы q.

Необходимый элемент произведения простых модулей находится путем домножения произведения $S = p_1 * p_2 * \ldots * p_{l_1-1}$ на первый непропущенный модуль p_t после l_1-го.

$S = S * p_t$.

Следующий элемент произведения простых модулей получается домножением S на следующий непропущенный модуль и т. д.

Псевдокод этого алгоритма будет иметь следующий вид.

Обозначим missingIndx[] – массив номеров исключенных модулей, rem[] – массив остатков. Тогда

```
BigInteger[] recoveryNewton(int[] missingIndx, long[] rem) {
        BigInteger[] res = new BigInteger[2];
        res[0] = BigInteger.valueOf(rem[0]);
        if(missingIndx != null){
           res[1] = BigInteger.valueOf
           (prims[missingIndx[0] +1]);
        } else {
           res[1] = BigInteger.valueOf(prims[0]);
        }
//если остаток один, то он и возвращается
        if (rem.length == 1) return res;
//Алгоритм Ньютона
        long resMod;
        if (missingIndx == null || missingIndx.length == 0) {
//в случае если пропусков нет, восстанавливаем число
только при помощи массивов
           for (i = 1; i < rem.length; i++) {
              resMod = res[0].mod(BigInteger.valueOf
                    (prims[i])). longValue();
              res[0] = res[0].add(BigInteger.valueOf((rem[i]
              – resMod)*inversMulPrm[i – 1]).
               multiply(mulPrm[i – 1]));
           }
           res[1] = mulPrm[rem.length – 1];
        } else {
//если пропуски есть, то до первого из них восстанавливаем число при помощи массивов
           for (i = 1; i < missingIndx[0]; i++) {
              resMod = res[0].mod(BigInteger.valueOf
                    (prims[i])).longValue();
              res[0] = res[0].add(BigInteger.valueOf((rem[i]
              – resMod)*inversMulPrm[i – 1]).
              multiply(mulPrm[i – 1])); }
//дальнейшие вычисления используют кэш частично
           int lengArr = missingIndx.length + rem.length;
           if(missingIndx[0] != 0)
             res[1] = mulPrm[missingIndx[0] – 1];
           BigInteger q, mulInvPrm;
           for (i = 0; i < missingIndx.length – 1; i++) {
              for (j = missingIndx[i] + 1;
              j < missingIndx[i + 1]; j++) {
//вычисление произведения "лишних" множителей в элементах массива mulInversPrm
                 q = BigInteger.ONE;
                 for (n = 0; n < i + 1; n++) {
                    q = q.multiply(BigInteger.valueOf
                    (NumberI. p_Inverse
                     (prims[missingIndx[n]], prims[j]))); }
//вычисление "реальных" элементов массива mulInversPrm и mulPrm
                 mulInvPrm = (mulInversPrm[j – 1].
                  divide(q)). mod(BigInteger.
                    valueOf(prims[j]));
                 resMod=res[0].
                    mod(BigInteger.valueOf(prims[j])).
                    longValue();
                 res[0] = res[0].add(BigInteger.valueOf
                    ((rem[j – i – 1] - resMod)).
                     multiply(mulInvPrm). multiply(res[1]));
                 res[1]=res[1].multiply(BigInteger.valueOf
                     (prims[j]));
              }
           }
//вычисление использующие данные о всех исключенных модулях
           int k = missingIndx[missingIndx.length – 1] + 1;
           for (i = k; i < lengArr; i++) {
//вычисление произведения "лишних" множителей в элементах массива mulInversPrm
              q = BigInteger.ONE;
              for (n = 0; n < missingIndx.length; n++) {
                 q = q.multiply(BigInteger.valueOf
                     (NumberI.p_Inverse(prims
                      [missingIndx[n]], prims[i]))); }
//вычисление "реальных" элементов массива mulInversPrm и mulPrm
              mulInvPrm = (mulInversPrm[i–1].divide(q)).
                     mod(BigInteger.valueOf(prims[i]));
               resMod=res[0].
                     mod(BigInteger.valueOf(prims[i])).
                     longValue();
              res[0] = res[0].add(BigInteger.valueOf
                  ((rem[i – missingIndx.length] – resMod)).
                   multiply(mulInvPrm) multiply(res[1]));
    res[1]=res[1].multiply(BigInteger.valueOf(prims[i]));
           }
        }
        res[0] = res[0].mod(res[1]);
        return res;
}
```

Рассмотрим задачу нахождения результанта полиномов от одной переменной с целыми коэффициентами, не превосходящими 65000, при помощи алгоритма Евклида с упрощением коэффициентов по методу Брауна [1]. Очевидно, что с ростом степени полиномов будет расти и величина результанта. Рассмотрим две метода: вычисление результанта с использованием чисел в формате BigInteger и модулярное вычисление результанта с применением препроцессинга, как изложено выше.

Эксперименты, проведенные в среде программирования Java для вышеуказанных полиномов, показали указанные в табл. 1 результаты по времени вычислений в мс.

Таблица 1

Степени полиномов	32	52	62	82	92	102	112
Время стандартного метода, мс	47	250	437	1125	1828	2641	3750
Время модулярного метода, мс	31	47	47	78	109	125	156
Отношение времен	1,5	5,3	9,3	14,4	16,77	21,1	24

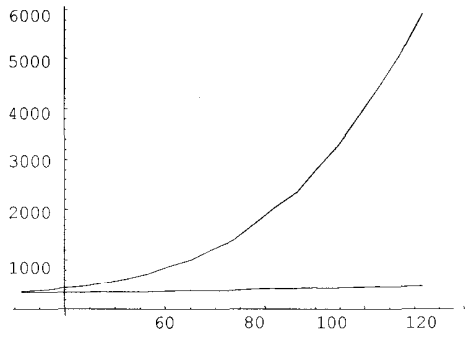

Рис. 1. Графики зависимости времени вычислений алгоритмов от порядков полиномов

Приведем графики зависимости времени вычислений для этих алгоритмов от порядков полиномов (рис. 1). Здесь виден существенный выигрыш модулярного алгоритма, превышающий теоретические оценки времени вычислений: стандартный алгоритм должен расти как пятая степень порядка полинома, а модулярный алгоритм – как четвертая степень.

ЛИТЕРАТУРА

1. *Brown W.S.* The Subresultant PRS Algorithm // ACM TOMS. 1978. № 4. P. 237-249.
2. *Кнут Д.Э.* Искусство программирования. Т. 2. Получисленные алгоритмы. М.: Издат. дом «Вильямс», 2001. С. 469-490.
3. *Малашонок Г.И.* Матричные методы вычислений в коммутативных кольцах. Тамбов: Изд-во ТГУ им. Г. Р. Державина, 2002. 214 с.

БЛАГОДАРНОСТИ: Работа выполняется при финансовой поддержке Российского Фонда фундаментальных исследований (грант №04-07-90268).

Поступила в редакцию 2 сентября 2006 г.