Efficient Evaluation of Large Polynomials

Charles E. Leiserson[1], Liyun Li[2], Marc Moreno Maza[2], and Yuzhen Xie[2]

[1] CSAIL, Massachussets Institute of Technology, Cambridge MA, USA
[2] Department of Computer Science, University of Western Ontario, London ON, Canada

Abstract. Minimizing the evaluation cost of a polynomial expression is a fundamental problem in computer science. We propose tools that, for a polynomial P given as the sum of its terms, compute a representation that permits a more efficient evaluation. Our algorithm runs in $d(nt)^{O(1)}$ bit operations plus $dt^{O(1)}$ operations in the base field where d, n and t are the total degree, number of variables and number of terms of P. Our experimental results show that our approach can handle much larger polynomials than other available software solutions. Moreover, our computed representation reduce the evaluation cost of P substantially.

Keywords: Multivariate polynomial evaluation, code optimization, Cilk++.

1 Introduction

If polynomials and matrices are the fundamental mathematical entities on which computer algebra algorithms operate, expression trees are the common data type that computer algebra systems use for all their symbolic objects. In MAPLE, by means of common subexpression elimination, an expression tree can be encoded as a directed acyclic graph (DAG) which can then be turned into a straight-line program (SLP), if required by the user. These two data-structures are well adapted when a polynomial (or a matrix depending on some variables) needs to be regarded as a function and evaluated at points which are not known in advance and whose coordinates may contain "symbolic expressions". This is a fundamental technique, for instance in the *Hensel-Newton lifting techniques* [6] which are used in many places in scientific computing.

In this work, we study and develop tools for manipulating polynomials as DAGs. The main goal is to be able to compute with polynomials that are far too large for being manipulated using standard encodings (such as lists of terms) and thus where the only hope is to represent them as DAGs. Our main tool is an algorithm that, for a polynomial P given as the sum its terms, computes a DAG representation which permits to evaluate P more efficiently in terms of work, data locality and parallelism. After introducing the related concepts in Section 2, this algorithm is presented in Section 3.

The initial motivation of this study arose from the following problem. Consider $a = a_m x^m + \cdots + a_1 x + a_0$ and $b = b_n x^n + \cdots + b_1 x + b_0$ two *generic* univariate polynomials of respective positive degrees m and n. Let $R(a, b)$ be the resultant of a and b. By generic polynomials, we mean here that $a_m, \ldots, a_1, a_0, b_n, \ldots, b_1, b_0$ are independent symbols. Suppose that $a_m, \ldots, a_1, a_0, b_n, \ldots, b_1, b_0$ are substituted to polynomials $\alpha_m, \ldots, \alpha_1, \alpha_0, \beta_n, \ldots, \beta_1, \beta_0$ in some other variables $c_1, \ldots, c_p$. Let us denote by $R(\alpha, \beta)$ the "specialized" resultant. If these α_i's and β_j's are large, then

K. Fukuda et al. (Eds.): ICMS 2010, LNCS 6327, pp. 342–353, 2010.

computing $R(\alpha, \beta)$ as a polynomial in $c_1, \ldots, c_p$, expressed as the sum of its terms, may become practically impossible. However, if $R(a, b)$ was originally computed as a DAG with $a_m, \ldots, a_1, a_0, b_n, \ldots, b_1, b_0$ as input and if the α_i's and β_j's are also given as DAGs with $c_1, \ldots, c_p$ as input, then one may still be able to manipulate $R(\alpha, \beta)$.

The techniques presented in this work do not make any assumptions about the input polynomials and, thus, they are not specific to resultant of generic polynomials. We simply use this example as an illustrative well-known problem in computer algebra.

Given an input polynomial expression, there are a number of approaches focusing on minimizing its size. Conventional common subexpression elimination techniques are typical methods to optimize an expression. However, as general-purpose applications, they are not suited for optimizing large polynomial expressions. In particular, they do not take full advantage of the algebraic properties of polynomials. Some researchers have developed special methods for making use of algebraic factorization in eliminating common subexpressions [1,7] but this is still not sufficient for minimizing the size of a polynomial expression. Indeed, such a polynomial may be irreducible. One economic and popular approach to reduce the size of polynomial expressions and facilitate their evaluation is the use of Horner's rule. This high-school trick for univariate polynomials has been extended to multivariate polynomials via different schemes [8,9,3,4]. However, it is difficult to compare these extensions and obtain an optimal scheme from any of them. Indeed, they all rely on selecting an appropriate ordering of the variables. Unfortunately, there are $n!$ possible orderings for n variables.

As shown in Section 4, our algorithm runs in polynomial time w.r.t. the number of variables, total degree and number of terms of the input polynomial expression. We have implemented our algorithm in the `Cilk++` concurrency platform. Our experimental results reported in Section 5 illustrate the effectiveness of our approach compared to other available software tools. For $2 \leq n, m \leq 7$, we have applied our techniques to the resultant $R(a, b)$ defined above. For $(n, m) = (7, 6)$, our optimized DAG representation can be evaluated sequentially 10 times faster than the input DAG representation. For that problem, none of code optimization software tools that we have tried produces a satisfactory result.

2 Syntactic Decomposition of a Polynomial

Let $\mathbb{K}$ be a field and let $x_1 > \cdots > x_n$ be n ordered variables, with $n \geq 1$. Define $X = \{x_1, \ldots, x_n\}$. We denote by $\mathbb{K}[X]$ the ring of polynomials with coefficients in $\mathbb{K}$ and with variables in X. For a non-zero polynomial $f \in \mathbb{K}[X]$, the set of its monomials is $\mathsf{mons}(f)$, thus f writes $f = \sum_{m \in \mathsf{mons}(f)} c_m\, m$, where, for all $m \in \mathsf{mons}(f)$, $c_m \in \mathbb{K}$ is the coefficient of f w.r.t. m. The set $\mathsf{terms}(f) = \{c_m\, m \mid m \in \mathsf{mons}(f)\}$ is the set of the terms of f. We use $\sharp\mathsf{terms}(f)$ to denote the number of terms in f.

Syntactic operations. Let $g, h \in \mathbb{K}[X]$. We say that gh is a *syntactic product*, and we write $g \odot h$, whenever $\sharp\mathsf{terms}(g\,h) = \sharp\mathsf{terms}(g) \cdot \sharp\mathsf{terms}(h)$ holds, that is, if no grouping of terms occurs when multiplying g and h. Similarly, we say that $g + h$ (resp. $g - h$) is a *syntactic sum* (resp. *syntactic difference*), written $g \oplus h$ (resp. $g \ominus h$), if we have $\sharp\mathsf{terms}(g+h) = \sharp\mathsf{terms}(g) + \sharp\mathsf{terms}(h)$ (resp. $\sharp\mathsf{terms}(g-h) = \sharp\mathsf{terms}(g) + \sharp\mathsf{terms}(h)$).