Sparse Polynomial Arithmetic Using Heaps of Pointers

Michael Monagan Roman Pearce

Introduction

Our goal is to compute with large sparse polynomials efficiently. The polynomials are multivariate and their terms are sorted in a *monomial order*. They may have coefficients in $\mathbb{Z}$ or in $\mathbb{Z}_p$. E.g.:

$$f = -9x^4 - 7x^3y + 6x^2y^3 + 8y^2$$

We store the polynomials as arrays, encoding small coefficients and exponents in place. Cache misses are reduced by eliminating pointers and random memory access. We can store f in 12 words:

coeff	*x*	*y*									
−9	4	0	−7	3	1	6	2	3	8	0	2

We pack multiple exponents into each word to reduce storage and speed up monomial operations. The savings are significant for problems in many variables, especially on 64-bit machines. Here is f with two exponents packed into each word:

coeff	*x*	*y*									
−9	4	0	−7	3	1	6	2	3	8	0	2

Classical Algorithms

To add or subtract two polynomials we use a merge, but adding multiple polynomials this way is slow. E.g., adding n polynomials with m terms each can produce intermediate sums with im terms. The cost of merging will be $\sum_{i=2}^{n}(im-1) \in O(n^2m)$. This can happen even when the result is small, from an *intermediate blowup* in the number of terms.

A divide-and-conquer approach is often used for multiplication to reduce the complexity to $O(nm \log n)$. With "geobuckets" the space required is $O(nm)$, the size of the result. Buckets with, e.g. $\{4, 8, 16, 32, \ldots\}$ terms are allocated and polynomials are merged into the first bucket that is larger. When buckets overflow their contents are merged into the next larger bucket.

Geobuckets are less attractive for exact division. If the quotient has n terms and the divisor has m terms, $O(nm)$ memory is used to cancel terms and produce zero when the result is $O(n)$ space. The complexity is $O(nm \log(nm))$, slower than a multiplication, due to computations of the buckets' leading term.

Heaps of Pointers

Instead of merging polynomials one by one into an intermediate object we can use a heap to do a simultaneous n-ary merge. The largest term of each polynomial is placed into the heap, and each time a term is extracted from the heap its successor is inserted. Summing n polynomials with m terms each is $O(nm \log n)$, the same complexity as divide-and-conquer. The heap requires $O(n)$ space plus storage for the result.

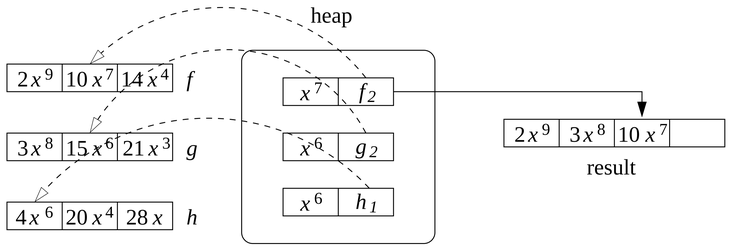

Figure 1: adding polynomials with a heap of pointers

To multiply fg we store two pointers in each element of the heap. The first points to a term f_i of f and the second pointer increments along g. The terms of each f_ig are constructed on-the-fly as they are inserted into the heap.

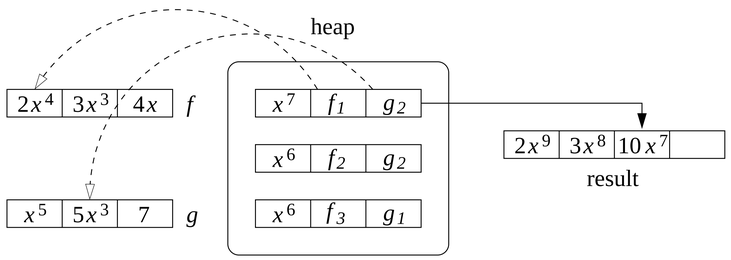

Figure 2: multiplying polynomials with a heap of pointers

Heap multiplications perform very well in the presence of cache. Let $\#f = n$ and $\#g = m$ with $n \leq m$. The computation of fg makes n simultaneous passes over g while randomly accessing f and a heap of size n. The heap and f are $O(n)$ space, and fit in the cache for practical values of n. Even if g does not fit in the cache, the memory access pattern will cache its terms effectively.

Division Using Heaps

To divide f by g we compute a quotient q while merging $f - qg$. Let $\#q = n$ and $\#g = m$ and $\#f \leq nm$. The straightforward approach is to merge f and each $-q_ig$ using a heap of size $n+1$. The terms of each $-q_ig$ are computed on-the-fly just like in the multiplication algorithm, but we must dynamically grow the heap because the size of the quotient is initially unknown.

This algorithm takes $O(nm \log n)$ time and uses $O(n)$ space, plus storage for the remainder if it is desired. One nice feature of using a heap is that terms are merged in descending order. For an exact division where division fails, we stop after merging the minimum number of terms.

An interesting alternative that is unique to heaps is the possibility to merge f with each $-g_iq$. That is, the heap elements point to terms of the divisor g and increment along the quotient q while q is being constructed.

This variant takes $O(nm \log m)$ time and uses $O(m+n)$ space, but only $O(m)$ memory is accessed randomly. The algorithm uses a heap of size m and does $m-1$ simultaneous passes over the quotient. For problems with small divisors and large quotients, this is substantially faster.

One such case is computing over algebraic number fields. The minimal polynomials of algebraic extensions are typically small, but they are frequently used to reduce terms, i.e., their quotients are large. With the second heap algorithm one can reduce large polynomials of very high degree in time and space that is linearly proportional to the number of reductions that occur.

References

[1] Stephen C. Johnson. Sparse polynomial arithmetic. ACM SIGSAM Bulletin, Volume 8, Issue 3 (1974) 63–71.

[2] Thomas Yan. The Geobucket Data Structure for Polynomials. *J. Symb. Comput.* **25** (1998) 285–293.

[3] Olaf Bachmann, Hans Schönemann. Monomial representations for Gröbner bases computations. *Proceedings of ISSAC 1998*, ACM Press (1998) 309–316.

[4] Michael Monagan and Roman Pearce. Polynomial Division using Dynamic Arrays, Heaps, and Packed Exponent Vectors. *Proceedings of CASC 2007*.

NSERC CRSNG

MITACS Symbolic Analysis Project – Supervisor Peter Borwein
Centre for Experimental and Constructive Mathematics
Computer Algebra Group
Simon Fraser University