Lazy Polynomial Arithmetic and Applications

Paul Vrbik
University of Western Ontario

July 8, 2009

Delayed / Lazy Computation

Lazy computation is an environment where calculations are made only when *absolutely* necessary.

Example

- The functional language Haskell is a “lazy” language which allows for the creation of infinite lists.
- Stephen Watt used delayed computation to work with power series in scratchpad.

How to make a polynomial lazy:

- Impose some ordering on the polynomial's terms.
- Only allow access to a single term of the polynomial.
- Do the as little work as possible to calculate that term.

$$
\begin{aligned}
f &= x^4y + x^2y^2 + 3 + 0 + 0 + \cdots \\
&= f_1 + f_2 + f_3 + f_4 + f_5 + \cdots \\
\Rightarrow \#f &= 3
\end{aligned}
$$

Remark

Our ordering is actually some monomial ordering $\succ$. When I say "largest term" or "in order" I mean the "$\succ$-largest term" or "$\succ$-order" (respectively).

What is the goal of lazy polynomial arithmetic?

- To calculate the n-th term of $f \times g$, $f + g$ or $f \div g$ using as *few* terms of f and g as possible.

Polynomial Multiplication

Classical Multiplication

$f \times g = ((f \times g_1 + f \times g_2) + f \times g_3) + \cdots + f \times g_m$ where additions are done using a simple merge (requires all of g!!).

Cost : $O(\#f\#g^2)$ $\succ$-comparisons for sparse polynomials.

Sort method

Sort $L = [f_1g_1, \ldots, f_ng_1, f_1g_2, \ldots, f_ng_2, \ldots, f_1g_m, \ldots, f_ng_m]$ and collect like terms.

Cost : Space to store $O(\#f\#g)$ terms.

Merge method

Do a simultaneous m-ary merge on the set of *sorted* sequences

$$S = \{(f_1g_1, \ldots, f_ng_1), \ldots, (f_1g_m, \ldots, f_ng_m)\}.$$

Heap Multiplication

Johnson's Heap Multiplication

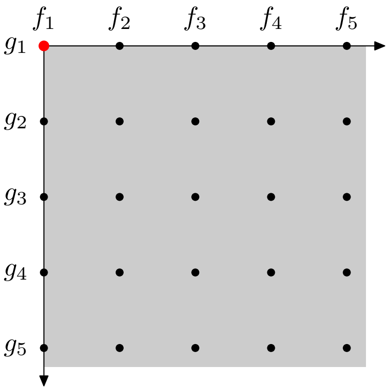
Use a heap, initialized to contain $f_1 g_1, f_1 g_2, \ldots, f_1 g_m$ to merge the m sequences (*still* uses all of g!).

Cost : $O(\#f \#g \log \#g)$ $\succ$-comparisons for sparse polynomials.

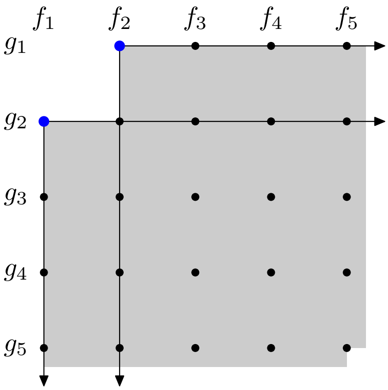
Our Heap Multiplication

Use a heap, initialized to contain f_1, and a replacement scheme to merge the m sequences.

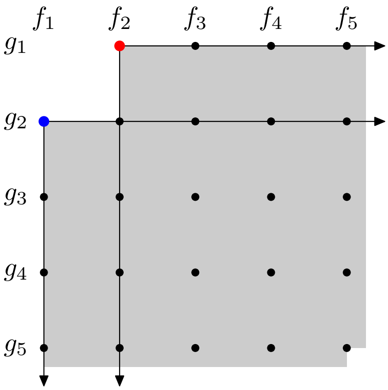
Heap Multiplication



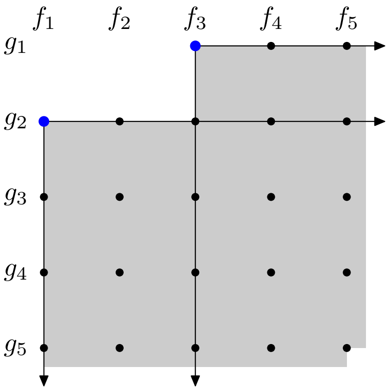
Heap Multiplication



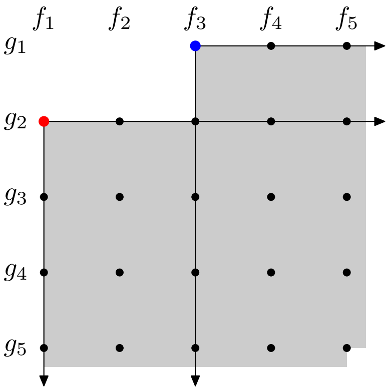
Heap Multiplication



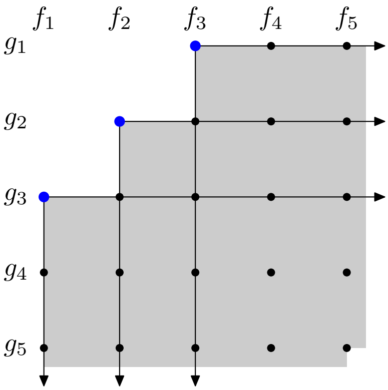
Heap Multiplication



Heap Multiplication



Heap Multiplication



Generalizing this idea we get a replacement scheme for the heap.

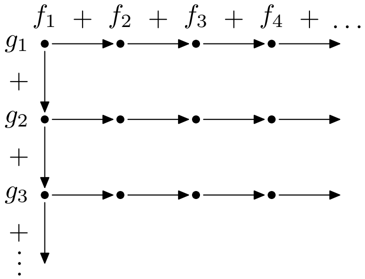


Figure: Points represent the terms of $f \times g$, arrows indicate the next $\succ$-largest term.

Generalizing this idea we get a replacement scheme for the heap.

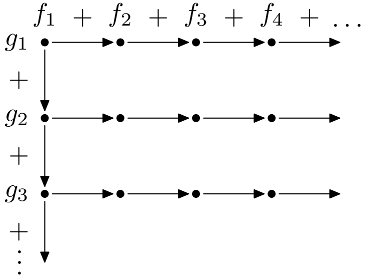


Figure: Points represent the terms of $f \times g$, arrows indicate the next $\succ$-largest term.

- Heap can only get as big as $O(\#g)$.

Generalizing this idea we get a replacement scheme for the heap.

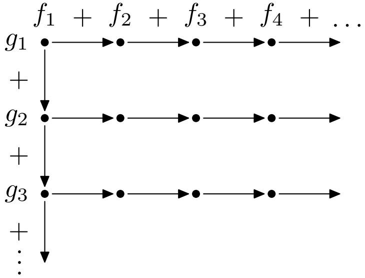


Figure: Points represent the terms of $f \times g$, arrows indicate the next $\succ$-largest term.

- Heap can only get as big as $O(\#g)$.
- Product has at most $\#f \cdot \#g$ terms.

Generalizing this idea we get a replacement scheme for the heap.

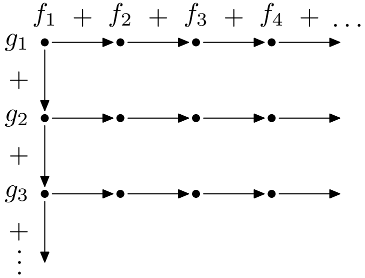


Figure: Points represent the terms of $f \times g$, arrows indicate the next $\succ$-largest term.

- Heap can only get as big as $O(\#g)$.
- Product has at most $\#f \cdot \#g$ terms.

$\Rightarrow$ Worst-case space complexity for heap multiplication is $O(\#f\#g + \#g)$.

Heap Division

For $f \div g$ construct the quotient q and remainder r such that $f - qg - r = 0$. We use a heap to store the sum $f - qg$ by merging the set of $\#q + 1$ sequences

$$\{(f_1, \ldots, f_n), (-q_1 g_1, \ldots, -q_k g_1), \ldots, (-q_1 g_m, \ldots, -q_k g_m)\}.$$

Alternatively we may see the heap as storing the sum

$$f - \sum_{i=1}^{m} g_i \times (q_1 + q_2 + \ldots + q_k)$$

where $\#g = m$, $\#q = k$ and the terms q_i may be unknown. That is, it possible that we remove $-q_{i-1} g_j$ before q_i is known, in which case we would sleep the term $-q_i g_j$.

Lazy Arithmetic

$H = \text{ADD}(F, G)$

- $O(\#f + \#g)$ monomial comparisons.
- Space complexity is $O(\#h)$.

$H = \text{MULT}(F, G)$

- $O(\#f\#g \log \#g)$monomial comparisons.
- Space complexity is $O(\#f\#g + \#g)$.

$Q, R = \text{DIVIDE}(F, G)$

- $O((\#f + \#q\#g) \log \#g))$ monomial comparisons.
- Space complexity is $O(1 + \#g + \#q + \#r)$.

Forgetful Polynomial

A forgetful polynomial is a variant of a lazy polynomial where calculated terms are *not* stored. That is, unlike lazy polynomials, we can not re-access terms.
Furthermore access is only given in $\succ$-order.

How to make a polynomial forgetful:

- Impose ordering ($\succ$) on the polynomial's terms
- Only allow access to single terms of the polynomial by way of a `next` command.

Forgetful Arithmetic

- The forgetful operations are different as they may or may not be able to return / accept forgetful polynomials.
- Full generalization of forgetful polynomial arithmetic is "impossible".

Why?

Regardless of the scheme used to calculate $f \times g$, it is necessary to multiply every term of f with g. Since we are limited to single time access to terms this task is impossible. If we calculate $f_1 g_2$ we can not calculate $f_2 g_1$ and vice versa.

Forgetful Arithmetic

$H = \mathrm{ADD}(F, G)$

- H, F, G can *all* be forgetful.
- Space complexity is $O(1)$.

$H = \mathrm{MULT}(F, G)$

- F and G can *not* be forgetful.
- H *can* be forgetful. (Important!)
- Space complexity is $O(\#g)$.

$Q, R = \mathrm{DIVIDE}(F, G)$

- G and Q can *not* be forgetful. $(F - Q \times G - R = 0)$.
- F, R *can* be forgetful. (Important!)
- Space complexity is $O(1 + \#g + \#q)$.

Why forget? Consider the division

$$\frac{A \cdot B - C \cdot D}{E} = Q \text{ with } R = 0.$$

Why store the sub-expression $A \cdot B - C \cdot D$ if you only care about Q?

Space complexity using the heap algorithms classically

$$O(\underbrace{\#A\#B + \#C\#D + \#B + \#D}_{\text{multiplication for dividend}} + \underbrace{\#E + \#Q}_{\text{division}})$$

Space complexity using forgetful algorithms

$$O(\underbrace{\#A + \#B + \#C + \#D + \#B + \#D}_{\text{multiplication for dividend}} + \underbrace{\#E + \#Q}_{\text{division}})$$

Bareiss' Algorithm for fraction free-determinant calculation.

Input: $\mathbf{M}$ an n-square matrix with entries in an integral domain $\mathcal{D}$.
Output: $\det(\mathbf{M})$.

```
M_{0,0} ← 1;
for k = 1 to n − 1 do
    for i = k + 1 to n do
        for j = k + 1 to n do
            M_{i,j} ← (M_{k,k} M_{i,j} − M_{i,k} M_{k,j}) / M_{k−1,k−1}   {Exact division.}
        end for
    end for
end for
return (M)_{n,n}
```

$$\mathbf{M}_{i,j} \leftarrow \frac{\mathbf{M}_{k,k}\mathbf{M}_{i,j} - \mathbf{M}_{i,k}\mathbf{M}_{k,j}}{\mathbf{M}_{k-1,k-1}} \quad \{\text{Exact division.}\}$$

Bariess' Algorithm Weaknesses

Let

$$A = \begin{bmatrix} x_1 & x_2 & x_3 & \cdots & x_9 \\ x_2 & x_1 & x_2 & \cdots & x_8 \\ x_3 & x_2 & x_1 & \cdots & x_7 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ x_9 & \cdots & x_3 & x_2 & x_1 \end{bmatrix}.$$

When calculating det(A) using Bareiss' algorithm the last division will have:

- A dividend of 128,530 terms.
- A divisor of 427 terms.
- A quotient of 6,090 terms (this is the determinant).

Let $Q = \frac{A \times B - C \times D}{E}$ be the division of line 5 of the Bareiss algorithm and $\alpha = \max(\#A, \#B) + \max(\#C, \#D)$. The following is a measurement of memory used by our implementation of the Bareiss algorithm using forgetful polynomials to calculate $\mathbf{M}_{n,n}$ when given the Toeplitz matrix generated by $[x_1, \ldots, x_7]$.

n	$\#A$	$\#B$	$\#C$	$\#D$	$\#E$	$\#A\#B + \#C\#D$	$\alpha + \#E + \#Q$
5	12	15	17	17	4	469	106
6	35	51	55	55	12	4810	306
7	35	62	70	70	12	7070	326
8	120	182	188	188	35	57184	832

For $n = 8$ the total space is reduced by a factor of $57184/832 = 68$ (compared to a Bareiss implementation that explicitly stores the quotient), which is significant.

Pseudo-remainders

For $f = 3x^3 + x^2 + x + 5, g = 5x^2 - 3x + 1 \in \mathbb{Z}[x]$, dividing f by g would produce the quotient and remainder

$$q = \frac{3}{5}x + \frac{14}{25} \quad \text{and} \quad r = \frac{52}{25}x + \frac{111}{25}.$$

Whereas, if we premultiplied f by 5^2 and divided $5^2 f$ by g we would get a pseudo-quotient and pseudo-remainder

$$\tilde{q} = 15x + 14 \quad \text{and} \quad \tilde{r} = 52x + 111.$$

Moreover, no fractions appear while executing the division algorithm thereby avoiding calculations in $\mathbb{Q}$.

The Extended Subresultant algorithm.

Input: The polynomials $u, v \in \mathcal{D}[x]$ where $\texttt{deg}_x(u) > \texttt{deg}_x(v)$.

Output: $r = \mathrm{Res}(u, v, x)$ and $s, t \in \mathcal{D}[x]$ satisfying
$s \cdot u + t \cdot v = \mathrm{Res}(u, v, x) \Rightarrow u^{-1} \equiv s/\mathrm{Res}(u, v, x) \mod v$ in $\mathcal{D}/\mathcal{D}[x]/v$.

$(g, h) \leftarrow (1, -1)$; $(s_0, s_1, t_0, t_1) \leftarrow (1, 0, 0, 1)$;
while $\texttt{deg}_x(v) \neq 0$ **do**
  $d \leftarrow \texttt{deg}_x(u) - \texttt{deg}_x(v)$;
  $\tilde{r} \leftarrow \texttt{prem}(u, v, x)$; $\{\tilde{r}$ is big.$\}$ $\tilde{q} \leftarrow \texttt{pquo}(u, v, x)$;
  $u \leftarrow v$; $\alpha \leftarrow \texttt{lcoeff}_x(v)^{d+1}$;
  $(s, t) \leftarrow (\alpha \cdot s_0 - s_1 \cdot \tilde{q}, \alpha \cdot t_0 - t_1 \cdot \tilde{q})$;
  $v \leftarrow \tilde{r} \div (-g \cdot h^d)$; {Exact division.}
  $(s_0, t_0) \leftarrow (s_1, t_1)$;
  $(s_1, t_1) \leftarrow (s \div (-g \cdot h^d), t \div (-g \cdot h^d))$;
  $g \leftarrow \texttt{lcoeff}_x(u)$;
  $h \leftarrow (-g)^d \div h^{d-1}$;
end while
$(r, s, t) \leftarrow (v, s_1, t_1)$;
return v, s_1, t_1;

Example

Consider the two polynomials;

$$f = x_1^6 + \sum_{i=1}^{8} (x_i + x_i^3)$$

$$g = x_1^4 + \sum_{i=1}^{8} x_i^2$$

$\mathbb{Z}[x_1, \ldots, x_9]$. When we apply the extended subresultant algorithm to these polynomials we find that in the last iteration, the pseudo-remainder $\tilde{r}$ has $427,477$ terms but the quotient v has only $15,071$ (v is the resultant in this case).

Let $\tilde{r}, \tilde{q}$ be from line 5 and $v, -g \cdot h^d$ be from line 10 of Algorithm 7. The following is a measurement of the memory used by our implementation of the extended subresultant algorithm using forgetful polynomials to calculate $\mathrm{Res}(f, g, x_1)$ where

$$f = x_1^8 + \sum_{i=1}^{5} \left(x_i + x_i^3\right), g = x_1^4 + \sum_{i=1}^{5} x_i^2 \in \mathbb{Z}[x_1, \ldots, x_5]$$

at iteration n.

n	$\#\tilde{r}$	$\#\tilde{q}$	$\#v$	$\#\left(-g \cdot h^d\right)$
1	29	7	29	1
2	108	6	108	1
3	634	57	634	1
4	14,692	2412	2,813	70

Implementation

- Implementation was done in C and then interfaced with Maple by way of a custom wrapper.
- Uses a "packed representation" for monomials which yields fast monomial comparisons and multiplications.

Benchmarks

Table: Benchmarks for Maple's SDMP package, Maple 11, and our Lazy package.

	$f \times g \mod 503$			$(fg) \div f \mod 503$		
	SDMP	Maple11	Lazy	SDMP	Maple11	Lazy
$f = (1 + x + y^3)^{100}$ $g = (1 + x^3 + y)^{100}$	0.5	12.3	4.9	0.6	18.3	4.9
$f = (1 + x + y^2 + z^3)^{20}$ $g = (1 + z + y^2 + x^3)^{20}$	0.26	6.26	1.2	0.28	12.6	1.4
$f = (1 + x + y^3 + z^5)^{20}$ $g = (1 + z + y^3 + x^5)^{20}$	0.35	8.19	1.3	0.38	12.6	1.4

data structure for lazy polynomial.

```
struct poly {
    int N;
    TermType *terms;

    struct poly *F1;
    struct poly *F2;
    TermType (*Method)(int n, struct poly *F,
                       struct poly *G, struct poly *H);

    int state[6];
    HeapType *Heap;
};

typedef struct poly PolyType;
```

```
TermType Term (int n, PolyType *F) {
    if (n>F->N) {
        return F->Method(n,F->F1,F->F2,F);
    }
    return F->terms[n];
};
```

This procedure would be invoked like this:

```
Term(1,F).mono;
Term(1,F).coeff;
```

Conclusion

Contributions:

- Development of the lazy / forgetful algorithms.
- Proofs for space complexities of lazy / forgetful algorithms.
- Reducing space complexity of Bareiss' algorithm from quadratic to linear.
- A subresultant algorithm where explicitly storing large pseudo-remainders is not necessary.
- High performance C-implementation of these ideas.

Thanks!