The Geobucket Data Structure for Polynomials*

Thomas Yan
Computer Science
Cornell University
tyan@cs.cornell.edu

September 13, 1996

Abstract

The *geobucket* data structure is suitable as an intermediate representation of polynomials for performing large numbers of polynomial additions and lead term extractions. A sum involving N terms has worst-case running time $O(N \log N)$ both online and offline, matching or surpassing the obvious/standard alternatives. This makes the geobucket a good choice for performing the reduction step of Gröbner basis computations.

1 Introduction

A standard representation of polynomials is a sorted list of terms; this structure supports linear-time binary addition of polynomials and constant-time extraction of lead terms. Although appealingly simple and adequate for small calculations, the list representation usually behaves poorly for large aggregate sums, such as those arising during Gröbner basis computations (non-commutative as well as commutative) [Mor94]. The inner loop of such computations reduces a polynomial p as follows:

```
while $p \neq 0$ and not done do
    Extract the lead term of $p$
    Perhaps add another polynomial summand to $p$
```

where the polynomial summands added to p depend on the extracted lead terms but are not known initially. A frequent feature of reductions is for partial sums (accumulated in p) to be(come) much larger than summands; to avoid poor performance in such situations, an appropriate representation must be chosen for partial sums.[1], [2], [3]

[*] Support for this work came from Army NDSEG fellowship DAAL03-90-G-0158, ONR grant N00014-92-J-1973, and NSF grant CCR-9503319.

[1] Growth of partial sums is unsurprising. Suppose that a partial sum has M terms and the next summand has m terms, of which k are deleted through cancelation. Then k terms are also deleted from the partial sum, yielding a new partial sum with $M + m - 2k$ terms. For the result to be no larger than M terms requires $k \geq m/2$, i.e. at least half the terms of the summand must cancel. Even if the final result is 0, as is common for reductions, it is unlikely that summands are arranged for so much cancelation to consistently occur.

[2] Large size differentials in the opposite direction generally are not a problem: computation of a large summand usually dominates the cost of computing the corresponding partial sum.

[3] We ignore the cost of computing summands and assume that they are list polynomials. The former generally takes time at least linear in the number of terms, regardless of representation. Since conversion to another representation typically takes at most linear time, it is asymptotically negligble. This situation certainly applies to reductions, where summands are obtained by scaling basis elements. Basis elements are naturally represented by list polynomials because their terms are obtained already sorted, and regardless of representation, scaling takes at least linear time. Thus, the pseudo-code above captures the asymptotic running time of reductions, modulo costs independent of representation.

In November 1992, using *geobuckets* (defined in Section 4) instead of lists for partial sums allowed us to compute a Gröbner basis for commuting 4-by-4 matrices (see Appendix C) at least 32 times faster — under 43 hours instead of over 8 weeks.[4] This dramatic speedup is due to better treatment of the large size differentials between summands and the corresponding partial sums: one reduction had 1.7×10^5 summands, with partial sums of at least 1.2×10^5 terms, but the average size of summands was only 46 terms. By adding only polynomials of comparably bounded size, the *geobucket* data structure achieves $O(N \log N)$ worst-case asymptotic running time for a sum containing N terms interspersed with up to N lead term extractions (each term being extracted at most once).

Binomial heaps [Koz92, Bro78] initially appear to be a good alternative because they can be *melded* (merged) quickly and support reasonably fast extraction of a minimal element. A key feature is that heap meld combines only heap-ordered trees of equal size, so it gracefully accommodates large size differentials. Binomial heaps do have the same asymptotic worst-case running time as geobuckets, but geobuckets are simpler to implement and in practice are likely to have superior time and space complexity, especially in the cases where lists would perform efficiently. As we saw from the example of 4-by-4 commuting matrices ("the 4-by-4 example", henceforth), this makes the geobucket a good choice for performing reductions in Gröbner basis computations.

The rest of the paper is organized as follows. Section 2 gives terminology and the cost model. Section 3 critiques the standard list approach. Section 4 defines the geobucket data structure. Section 5 analyzes its worst-case time and space complexity. Section 6 presents empirical results and conclusions. Appendices give geobucket variations, other approaches, and specify benchmarks.

2 Preliminaries

Fix a set of *coefficients* that is closed under addition and has an identity 0 (zero). Fix a set of *monomials* disjoint from the coefficients, and fix a total *monomial order.* Note: We have omitted details irrelevant to our data structure.[5]

A *term* $a \cdot \alpha$ is a product of a non-zero coefficient a and a monomial α; *like* terms have the same monomial. The monomial order induces a partial order on terms — like terms are incomparable.

A *polynomial* is a finite sum of non-like terms. The *zero* polynomial is the empty sum, with no terms. The *size* $\#f$ of a polynomial f is the number of terms in the sum. The largest term $a \cdot \alpha$ of a non-zero polynomial f is called its *lead term* $\mathrm{LT}(f)$; a and α are called its *lead coefficient* $\mathrm{LC}(f)$ and *lead monomial* $\mathrm{LM}(f)$, respectively. *(Lead term) extraction* is the process of removing the lead term from f, leaving behind the polynomial $\mathrm{tail}(f) = f - \mathrm{LT}(f)$ consisting of the remaining terms.

Note: for polynomials, "large" and "small" refer solely to size and never to lead terms.

Polynomial *addition* is performed as expected: like terms are combined by summing their coefficients, and resulting terms with zero coefficients are deleted. We refer to the combining of like terms as *cancelation*, even if the coefficients do not sum to zero, because combining eliminates at least one term from the sum.

We are interested in the time and space complexity of adding n polynomials $f_1, \ldots, f_n$ interspersed with up to N extractions performed on the partial sums, where $N = \#f_1 + \cdots + \#f_n$ is the

[4] With some guidance from Mike Stillman, we did the computation attributed to him in [Hre94], Remark 2; we were perhaps the first to compute such a basis. The 8 week estimate is conservatively extrapolated from current runs: 15.3 elapsed (wall-clock) hours versus over 3 cpu weeks.

[5] Coefficients usually form a field or at least a ring. Monomials usually are all finite products over a fixed, finite set of variables, but in the non-commutative case, monomials are formed differently and monomial multiplication is non-commutative. Gröbner basis computations impose additional requirements, e.g. *strong computability* of the coefficients [Mis93] and *admissability* of the monomial order [Mor94].

total number of terms.[6] Operations are online, i.e. operations and operands are not known ahead of time. To simplify the analysis, assume all f_i's are non-zero. By this assumption, $n \le N$, so n is $O(N)$. If the actual input can contain zeros, then these can be filtered out in $O(n)$ additional time using no additional space.

We measure space in the number of terms and time in the number of monomial operations, primarily monomial comparisons. We ignore the number of coefficient additions because it is dominated by the number of monomial operations and is the same, up to a constant factor, for all algorithms considered.

3 List Polynomials

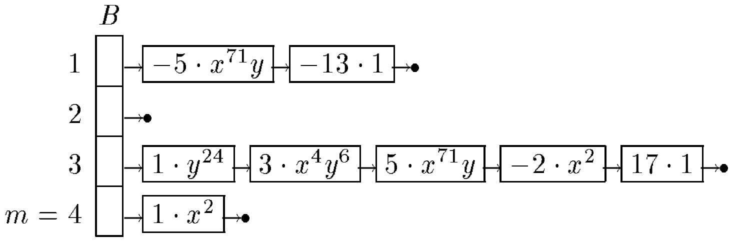
One standard representation of a polynomial is a *list* polynomial: a list of terms, sorted in decreasing order. Lead term extraction takes constant time because the lead term is at the front of the list. Binary addition $f_1 + f_2$ of two polynomials is done by merging the two sorted lists and canceling like terms. Space usage is optimal since cancelation eliminates storage of unnecessary terms. Observe that each comparison in the merge either places the smaller of two non-like terms or places the sum of both like terms into the result. Thus, the number of monomial comparisons performed during addition can be as small as $\min(\#f_1, \#f_2)$, when all the terms of the smaller summand precede or cancel with the lead terms of the other, or as big as $\#f_1 + \#f_2 - 1$, when the terms from the summands are non-trivially interleaved (-1 because when only the last term remains, there is no need for comparison).

This extremal analysis of binary addition extends easily to aggregate sums $f_1 + \cdots + f_n$ by considering the cost of adding the next summand f_i into the current partial sum. In the best case, addition of f_i takes at most $\#f_i$ comparisons, for a total of at most N comparisons.[7] In the worst case, with no cancelation and each summand of size $\#f_i = 1$, the i-th partial sum has $i-1$ terms, so addition of f_i takes $(i-1)+\#f_i-1 = i-1$ comparisons, for a total of $0+1+\cdots+(N-1) \simeq N^2/2$ comparisons. This clearly shows the potential problem that large size differentials can cause: deep penetration of a large partial sum by a small summand is expensive. In the best-case scenario, there can be a large size differential, but penetration is shallow relative to the size of the larger polynomial. The poor experimental behavior of list polynomials — e.g. in the 4-by-4 example (see Section 6) — shows that penetration tends to be deep, so large size differentials must be dealt with to avoid $O(N^2)$ behavior.

The worst-case behavior is analogous to that of insertion sort, which consistently merges singleton lists into large lists. Geobuckets are partly motivated by the superior performance of merge sort, which merges only lists of comparable size. (In the worst-case scenario above, the pattern of comparisons performed by geobuckets is identical to that of merge sort.) This is achieved by regrouping known summands. To see how this pays off, consider the worst-case cost of computing a sum $f_1 + f_2 + f_3$ with $\#f_1 \gg \#f_2 = \#f_3 = 1$. Computing left-to-right takes $2\#f_1 + 1$ comparisons, but right-to-left takes only $\#f_1 + 2$ comparisons: after f_2 and f_3 are cheaply combined, the sorted terms in their sum share the cost of penetrating f_1 in a single pass. Thus, the key to geobuckets is (re)grouping summands.

[6] Spatial locality of memory accesses is important for large calculations. Although our analysis and experimental data do not address this, geobuckets should have good spatial locality because they are built upon list polynomials, which can be implemented as arrays, and because the list polynomials are sequentially accessed and are fairly large for the most part.

[7] We can achieve any fraction of N comparisons by making the last summand sufficiently large.



$$
\begin{aligned}
&= (-5 \cdot x^{71}y - 13 \cdot 1) + 0 + (1 \cdot y^{24} + 3 \cdot x^4y^6 + 5 \cdot x^{71}y - 2 \cdot x^2 + 17 \cdot 1) + (1 \cdot x^2) \\
&= y^{24} + 3 \cdot x^4y^6 + (-5+5) \cdot x^{71}y + (-2+1) \cdot x^2 + (-13+17) \\
&= y^{24} + 3 \cdot x^4y^6 - x^2 + 4
\end{aligned}
$$

Figure 1: Sample geobucket ($d = 2$, $m = 4$). Coefficients are integers, and monomials x^iy^j are ordered by $x^iy^j > x^{i'}y^{j'}$ if $j > j'$ or both $j = j'$ and $i > i'$.

4 Geobucket Data Structure and Operations

We define the geobucket and give implementations for creation, addition, canonicalization, and lead term extraction, which simultaneously tests for zero. See Figure 2 for pseudo-code.

Fix a constant d, $1 < d \in \mathbb{R}$; a good choice is determined in Section 5. For the rest of the paper, logarithms $\log(\cdot)$ are base d.

Definition 1 *A* geobucket *is an array $B[1..m]$ of* buckets *$B[i]$ of* capacity *d^i: bucket $B[i]$ is a list polynomial of size $\#(B[i]) \le d^i$. The* canonical polynomial *represented by geobucket B is the sum $B[1] + \cdots + B[m]$ of its buckets.*

See Figure 1 for an example and note the following: there is some *internal cancelation* of terms; bucket $B[1]$ is full with $d^1 = 2^1 = 2$ terms; bucket $B[2]$ is empty; bucket $B[3]$ is only partly full, but its contents of 5 terms exceeds the capacity $d^2 = 2^2 = 4$ of $B[2]$; and bucket $B[4]$ is way under capacity.

To create the resizable array $B[1..m]$ of the buckets, allocate enough room from the start and maintain the number m of buckets in use. Below we show that $m \le \lceil \log N \rceil$. Since N is bounded by the amount of memory available and at most $\lceil \log N \rceil$ buckets are needed (see below), one can effectively compute a small constant upper bound on the space required. For example, given $2^{32-2} = 2^{30}$ addressable words, if $d = 4$, then at most $1 + \log 2^{30} = 16$ buckets are needed.[8]

To add a list polynomial f to geobucket B, add it into the smallest bucket allowed. The size of f and the contents of the bucket are both bounded by the bucket's capacity, so in this way only polynomials of comparably bounded size are added. Polynomials are not necessarily of comparable size because the contents of a bucket can be very small, e.g. zero. When the contents of a bucket exceeds its capacity, it *overflows*: its contents are added into the next bucket, which may also overflow, etc.; a sequence of overflows is called a *cascade*. Note that when the code shown in Figure 2 adds a polynomial f into a bucket $B[i]$ — both during the initial addition into a bucket and during cascades — the following condition holds:

$$\#f > d^{i-1} \text{ and } \#(B[i]) \le d^i. \tag{1}$$

[8] This approach treats N as bounded, which violates the spirit of asymptotic analysis and can waste excessive space if numerous small geobuckets are in use. These are unlikely to be practical concerns: $\log N$ is small, and in Gröbner basis computations, reductions are done sequentially, so it suffices to pre-allocate a single geobucket. However, see *array doubling* in Appendix A for a simple, general, standard solution.

Creation of $B = 0$

```
Allocate resizable array $B$ with entries initialized to 0
$m$ := 0
```

Addition of $B[1..m]$ with disposable list polynomial f

```
$i$ := max(1, $\lceil \log(\#f) \rceil$)
if $i \leq m$ then
    $f$ := $f + B[i]$
    while $i \leq m$ and $\#f > d^i$ do $f$ := $f + B[i+1]$; $B[i]$ := 0; $i$ := $i + 1$
$m$ := max($m$, $i$)
$B[i]$ := $f$
```

Canonicalization of $B[1..m]$ into list polynomial f

```
$f$ := 0
for $i$ := 1 to $m$ do $f$ := $f + B[i]$
```

Extraction of Lead Term from $B[1..m]$

```
repeat
    $j$ := 0
    for $i$ := 1 to $m$ do if $B[i] \neq 0$ then
        if $j = 0$ or $\mathrm{LM}[i] > \mathrm{LM}[j]$ then $j$ := $i$
        else if $\mathrm{LM}[i] = \mathrm{LM}[j]$ then $\mathrm{LC}[j]$ := $\mathrm{LC}[j] + \mathrm{LC}[i]$; $B[i]$ := $\mathrm{tail}[i]$
until $j = 0$ or $\mathrm{LC}[j] \neq 0$
if $j = 0$ then there is no lead term because $B = 0$
else lt := $\mathrm{LT}[j]$; $B[j]$ := $\mathrm{tail}[j]$; return lt
```

where $\mathrm{LT}[i] = \mathrm{LT}(B[i])$, $\mathrm{LM}[i] = \mathrm{LM}(B[i])$, $\mathrm{LC}[i] = \mathrm{LC}(B[i])$, and $\mathrm{tail}[i] = \mathrm{tail}(B[i])$.

Figure 2: Geobucket Code

New buckets are created only when f exceeds the capacity of the largest bucket. Since a bucket capable of holding all $N \leq d^{\lceil \log N \rceil}$ terms cannot have its capacity exceeded, the largest bucket ever used has index at most $\lceil \log N \rceil$, i.e.:

$$m \leq \lceil \log N \rceil < 1 + \log N. \tag{2}$$

To obtain the canonical polynomial represented by B, return the sum $B[1] + \cdots + B[m]$ of the buckets, computed from small to large buckets; this order is important (see Section 5). This summing approach is asymptotically faster than repeated extraction when extraction is done naively as shown, and it avoids the bookkeeping overhead of *incremental extraction* (see Appendix A).

To extract the lead term, linearly search through the lead terms of non-empty buckets, canceling like terms as they are discovered, so at least one term is always removed from a non-empty geobucket. Note that even if cancelation yields a zero coefficient, the code in Figure 2 searches through remaining buckets in case there are more like terms or there is a larger term.[9] If the coefficient is 0 at the end of searching all buckets, then — and only then — is the search repeated.

[9] This is somewhat arbitrary, as is also the decision where to accumulate coefficients during cancelation. Our choice is intended to simplify presentation of incremental extraction in Appendix A.

Discussion

Geobuckets are intended mainly for multivariate polynomials. In the univariate case, polynomials tend to be(come) dense and consequently are best represented as arrays indexed by degree. This is true of the partial sums in Gröbner basis reductions and also of the overall Gröbner basis computation, which reduces to Euclidean GCD.

A more general setting for the geobucket data structure would be priority queues with deletion/cancelation of duplicates, but we prefer to deal with polynomials, since they are our primary application and we cannot think of natural examples where most priority queues (polynomial summands in our case) are already sorted. Also, cancelation is an important feature; this is evident from our empirical data (see Section 6), which shows that binomial heaps perform poorly due to their failure to perform cancelation.

The geobucket technique avoids adding small summands to large partial sums. The name *geobucket* comes from the implementation of this strategy: a partial sum is split among buckets whose capacities form a geometric series. Addition is performed only in suitably accommodating buckets and therefore only between polynomials of comparably bounded size. Extraction is performed as a linear search through buckets. This arrangement is reminiscent of other data structures and algorithms, such as the priority queue (see Appendix B). Another example is binomial heaps [Koz92, Bro78], which are composed of geometrically-sized heap-ordered trees. Also, Brown mentions that for various tree structures, a series of insertions can be sped up to achieve the same asymptotic complexity as for binomial heaps by using a collection of geometrically sized trees ([Bro78], page 304). Finally, cascades resemble and were inspired by the *array doubling* technique in Appendix A, which presents some variations on the geobucket data structure.

5 Worst-Case Analysis

Theorem 1 (Space Complexity) *In the worst case, $O(N)$ space is used, even if only $O(1)$ space is required for the list representation.*

This is easily proven. Let $f_1, \ldots, f_{n-1}$ be polynomials of sizes $d, d^2, \ldots, d^{n-1}$ with distinct monomials. Set $f_n = -(f_1 + \ldots + f_{n-1})$. Summing these places each f_i into a distinct bucket, so no cascades and hence no cancelation is performed, although the canonical polynomial is clearly 0.

For time complexity, the paramount cost is from addition. We charge the cost to terms of summands and overflows — to the list polynomials f added into buckets $B[i]$, not to the terms in the buckets $B[i]$.

Lemma 2 *Each term can be added into at most $1 + \log N$ buckets.*

Since cascades move terms only from smaller to larger buckets, a term can be added to each bucket at most once, which by (2) is at most $1 + \log N$. This can be thought of as the initial addition plus possibly cascading up through an additional $\log N$ buckets.

Lemma 3 *The cost $(\#f + \#(B[i]))/\#f$ per term in f of adding a list polynomial f into a bucket $B[i]$ is at most $1 + d$ comparisons.*

Recall that (1) holds, reproduced here for convenience:

$$\#f > d^{i-1} \text{ and } \#(B[i]) \leq d^i. \tag{1}$$

Since the bucket can have at most a factor d more terms than f, the cost per term is at most $1 + d$.

Theorem 4 (Cost of addition) *The total number of monomial comparisons incurred by additions is at most* $(d+1)(1+\log N)N$.

This is simply the total for adding all terms into as many buckets as possible, given the bounds from Lemmas 2 and 3.

Theorem 5 (Cost of Canonicalization) *The cost of canonicalization is at most* N *monomial comparisons beyond the cost of addition.*

Canonicalization is basically a forced cascade of all buckets — the first conjunct of (1) can be violated. The contents of bucket $B[m-i]$ are added at most $i+1$ times, so each term is compared at most $i+1$ times, but this is already (mostly) accounted for! The analysis of addition provides for moving a term in bucket $B[m-i]$ by i places up to bucket $B[m]$ and performing $d+1>2$ comparisons at each step along the way. This yields $(d+1)i=0$ when $i=0$, i.e. this does not cover the cost of inspecting terms in the largest bucket $B[m-0]$. But for all other buckets, where $i \geq 1$, this more than covers the $i+1 \leq 2i < (d+1)i$ comparisons required for canonicalization, and helps to defray costs for $B[m]$. That is, the cost of canonicalization is accounted for, except for possibly the cost of inspecting the largest bucket, which has at most N terms.

Theorem 6 (Cost of Lead Term Extraction) *At most* $N\log N$ *comparisons are performed for lead term extraction.*

Since there are at most $1+\log N$ buckets, a linear search performs at most $\log N$ comparisons. Each linear search of a non-empty geobucket removes at least one term, so at most N non-trivial linear searches can be performed, for a maximum number of comparisons of at most $N\log N$.

Corollary 7 (Total Cost) *The number of monomial comparisons incurred from performing addition, interspersed with up to* N *lead term extractions and terminated by at most one canonicalization, is at most*

$$(d+1)(1+\log N)N + N\log N + N = (d+2)N\log N + (d+2)N$$

Corollary 8 (Time Complexity) *The worst-case asymptotic time complexity is* $O(N\log N)$.

During lead term extraction and list polynomial addition, each coefficient sum is preceded by a monomial comparison. Creating and resizing the resizable array are constant time operations. The linear cost of list polynomial addition dominates the cost of array access — both the cost of sequential access and the cost of computing $\lceil\log(x)\rceil$. Thus, monomial comparisons asympotically dominate all other costs.

Choice of d

When N is large, the worst-case cost as measured by monomial comparisons is dominated by the term $(d+2)N\log N$, which is basically the cost of cascades. Applying the identity $\log x = \ln x/\ln d$ and pulling out the $N\ln N$ factor leaves us with $(d+2)/\ln d$ to minimize. The minimum occurs at

$$d \simeq 4.3.$$

If we attempt to minimize just the cost of cascades, then we get

$$d \simeq 3.6.$$

Empirically searching for d as a power of 2, we found that $d = 4$ worked best for the 4-by-4 example; this is a gratifying correspondence between an actual run and our worst-case analysis.

This choice of d strikes a balance between two extremes. If d is too large, then $O(d^2)$ behavior within buckets can be a problem. But our choice of d is relatively small. If d is too small, then cascades happen too frequently, incurring lots of list polynomial additions. But this is addressed by our explicit minimization of the overall cost of cascades.

Miscellaneous Remarks

We do not know how to characterize space usage any better than our coarse analysis. We suspect that during Gröbner basis computations there is often a substantial amount of internal cancelation since reductions are often 0. However, as redundant terms accumulate, cascades probably become both more likely to occur and also more likely to eliminate internal cancelation when they do occur.

The bounds of Lemmas 2 and 3 are overly pessimistic for the following three reasons. First, buckets often fill up gradually, so the average cost per term of addition is closer to $(d + 1)/2$ than to $d + 1$. Secondly, cancelation and extraction eliminate terms, further lowering the cost. Finally, the terms of a large summand do not move up from the bottom bucket $B[1]$. Thus, actual costs should be much lower, and even our worst-case analysis is overly pessimistic by a factor of about 2.

Although canonicalization can incur at most N more comparisons, these N can account for the bulk of comparisons. If we overestimate the size of the contents of bucket $B[m - i]$ as N/d^i, then the total number of comparisons performed is at most

$$N\left(1 + \frac{2}{d} + \frac{3}{d^2} + \cdots\right) = N\left(1 + \frac{1}{d} + \frac{1}{d^2} + \cdots\right)^2 = N\frac{d^2}{(d-1)^2}.$$

For $d = 4$, this gives $1.78N$ comparisons, so over half can come from the largest bucket.

Something interesting happens to cascades if d is small. Consider how big the overflow from buckets below $B[i]$ can get:

$$d + d^2 + \cdots + d^{i-1} = \frac{d^i - d}{d - 1}$$

Note that d^i becomes much larger than d, e.g. $4^8 = 65536$ versus 4, so we can approximate the numerator by d^i. If $d \geq 2$, then the denominator $d - 1 \geq 1$ keeps the overflow below the capacity d^i of $B[i]$. But if $d < 2$ is sufficiently small, then the denominator $d - 1 < 1$ allows the overflow to exceed the capacity of $B[i]$. So, strictly speaking, the line m `:= max(`m`,` i`)` in Figure 2 should be immediately preceded by the line i `:= max(`i`,` $\lceil \log(\#f) \rceil$`)`.

6 Empirical Results

We computed various Gröbner bases (see Table 1 for profiles) using different polynomial representations and collected time and space data: total number of monomial comparisons (Table 2), total time spent on polynomial manipulation (Table 3), cumulative space usage (Table 4), and amount of penetration (part of Table 1). We now give more detailed descriptions and explain how data were collected.

6.1 Explanation of Data

The sample runs were homogenized cyclic roots of unity of degrees 5 through 7 ($u5$, $u6$, $u7$) and runs of the 4-by-4 example truncated to degrees 5 through 8 ($f5$, $f6$, $f7$, $f8$). See Appendix C for definitions. For each benchmark, we computed the total number $\sum N$ of terms, the number $\sum n$ of summands, the average size $\sum N / \sum n$ of summands, total number of reductions, the maximum number N of terms in any reduction and the corresponding number n of summands, and an estimate of penetration (see below). See Table 1.

Of the alternative representations, binomial heaps looked the most promising; futhermore, we found that binomials heaps benefited from aggressive cancelation of like terms as they are discovered. We therefore tried the following representations: lists (L), geobuckets (G), binomial heaps of polynomials (H'_P) with aggressive cancelation, and binomial heaps of terms with aggressive cancelation (H') and without (H). See Appendix B for a discussion of binomial heap variants and other alternatives.

Monomial comparisons are counted using a global counter.

We estimate the time spent on polynomial operations as follows. Consider the total time as the sum $O + P$ of the time P spent on polynomial operations and the time O spent doing everything else. Given O, we can subtract to obtain P. We therefore need to estimate the overhead O. We do this by obtaining the time $O + L$ and the time $O + G$ for lists and geobuckets, respectively. Next, we perform the same computation on a list copy and also a geobucket copy of the same data structure. This takes time at least $O + L + G$. The difference $(O + L) + (O + G) - (O + L + G)$ gives us an upper bound on O, which we subtract from total times to obtain the estimated time spent on polynomial operations.

The approach above underestimates O, since the combined run has poorer cache behavior and higher runtime costs, most notably garbage collection. Indeed, for $f8$, this approach yields a negative estimated overhead, so the estimated overhead for $f7$ is used instead (see Table 3). But this should suffice to obtain a closer measure of time spent on polynomial operations. All runs were on a Sun 4/670 under Solaris 2.5 with 256Mb of real memory, so paging was not a major concern, but repeated runs indicate that times are accurate to about only two significant digits anyway.

Fix a representation and let $\#p_i$ be the number of terms used to represent the i-th partial sum $p_i = \sum_{j=1}^{i} f_j$. We measure cumulative space usage as the cumulative number $S = \sum_{i=1}^{n} \#p_i$ of terms surviving each addition. This gives more weight to larger polynomials: we don't much care if very small polynomials require an order of magnitude more space, but we care more if large polynomials require twice as much space. Also, large polynomials "last" longer in the sense that manipulating them requires more monomial operations and hence more time. The ratio of the cumulative space used by two different representations is a weighted average showing how much more/less space is used.

When using list polynomials for partial sums, we measure how deep penetration is during addition by expressing the actual number C of comparisons as a percentage of the number W of comparisons theoretically performed in the worst case. To estimate W, observe that in the worst case, addition inspects all terms. The resulting comparisons are those performed on terms surviving each addition — i.e. cumulative space S — and those performed on terms canceling to 0 — at most $N/2$ comparisons because each cancelation/comparison deletes 2 terms. Since S and $S + N/2$ bound W, we can bound the amount C/W of penetration by $C/(S + N/2)$ and C/S. We bound instead of measuring W because the lower bounds obtained already exhibit substantial penetration.

6.2 Conclusions

Table 2 shows the total number of monomial comparisons. Geobuckets and heaps require much fewer comparisons than lists for large runs, and geobuckets perform better than heaps. Also, heaps of list polynomials perform dramatically better than heaps of terms — it pays to use the inherent order present in summands. Furthermore, cancelation is so frequent that aggressive cancelation

	no.	ave.			max sum		
GB	red's	$\#f$	ΣN	Σn	N	n	penetration
$u5$	114	12	19435	1595	460	34	77% – 97%
$u6$	389	26	432756	16408	3454	109	73% – 89%
$u7$	2220	107	58494253	544550	146213	986	75% – 87%
$f5$	981	20	579311	29554	3456	226	60% – 63%
$f6$	1982	25	3900731	156736	25691	1173	71% – 73%
$f7$	3199	29	17601842	598145	99399	3838	75% – 75%
$f8$	4057	35	48500006	1403807	314996	10207	75% – 75%

Table 1: Profiles of Benchmarks

	$u5$	$u6$	$u7$	$f5$	$f6$	$f7$	$f8$
G	0.035	0.78	115	1.82	16.1	89	277
L	0.037 (1.0)	0.90 (1.2)	164 (1.4)	4.43 (2.4)	79.5 (4.9)	800 (9.0)	4330 (15.6)
H'_P	0.056 (1.6)	1.32 (1.7)	188 (1.6)	3.84 (2.1)	33.2 (2.1)	180 (2.0)	552 (2.0)
H'	0.108 (3.1)	2.61 (3.4)	378 (3.3)	6.91 (3.8)	56.8 (3.5)	295 (3.3)	894 (3.2)
H	0.168 (4.8)	5.21 (6.7)	1101 (9.5)	6.67 (3.7)	56.3 (3.5)	299 (3.4)	929 (3.4)

Table 2: Monomial comparisons (millions). The parenthesized numbers are the corresponding ratios with respect to geobuckets.

	$u5$	$u6$	$u7$	$f5$	$f6$	$f7$	$f8$
G	0.18	5.2	1016	12.2	132	907	3368
L	0.15 (0.8)	3.6 (0.7)	1100 (1.1)	19.7 (1.6)	376 (2.8)	4481 (4.9)	29523 (8.8)
H'_P	0.44 (2.4)	11.7 (2.2)	2546 (2.5)	30.4 (2.5)	286 (2.2)	1713 (1.9)	6774 (2.0)
H'	0.82 (4.6)	19.9 (3.8)	4752 (4.7)	51.3 (4.2)	431 (3.3)	2758 (3.0)	9507 (2.8)
H	1.24 (6.9)	41.3 (7.9)	10891 (10.7)	49.2 (4.0)	453 (3.4)	2896 (3.2)	10393 (3.1)
O	0.54	9.8	1084	19.4	97	235	235*

Table 3: Estimated time (seconds) for polynomial operations. The parenthesized numbers are the corresponding ratios with respect to geobuckets. At the bottom is a line O of estimated overheads. The estimated overhead for $f7$ is used for $f8$; see the text for an explanation.

	$u5$	$u6$	$u7$	$f5$	$f6$	$f7$	$f8$
L	0.038	1.0	189	7.1	109	1059	5781
G	0.053 [1.4]	1.5 [1.4]	374 [2.0]	10.6 [1.5]	170 [1.6]	1756 [1.7]	9751 [1.7]
H'_P, H', H	0.184 [4.9]	10.7 [10.6]	8974 [47.4]	17.6 [2.5]	308 [2.8]	3622 [3.4]	22888 [4.0]

Table 4: Cumulative space usage (millions of terms). The bracketed numbers are the corresponding ratios with respect to lists — not geobuckets. A single row for heaps suffices, since all variations store the same number of terms.

can yield significant improvement without much risk of degradation.

All of the preceding is also evident in Table 3, which shows total estimated time spent on polynomial operations. Observe that lists perform faster than geobuckets on $u5$ and $u6$. The superior performance of geobuckets over lists shows that there are indeed large size differentials and deep penetration. The latter is explicit in Table 1, which shows penetration when using lists of at least 60% in all benchmarks.

The ratios for time and the ratios for number of comparisons are somewhat related, validating our decision to count monomial operations as a measure of time. The correspondence is quite good for heaps of terms; interestingly, for lists, the time ratio is roughly half the comparison ratio. These correspondences are surprisingly good in light of other operations besides comparisons, e.g. coefficient arithmetic and inspection of empty buckets during lead term extraction.

Far from using arbitrarily more space, geobuckets typically use about 70% extra space in our benchmarks (Table 4). Heaps, however, perform much worse, using almost 50 times extra space for $u7$. This supports our contention that geobuckets are likely to use less space than heaps, especially in the cases that lists work well. *Periodic canonicalization* should improve this situation; see Appendix A for this and other geobucket variants.

The performance of heaps can probably be improved through periodic canonicalization and even more aggressive canceling. But these further complications are non-standard versions of binomial heaps, which are already non-trivial to implement. The effort involved is better spent improving geobuckets or lists, since geobuckets use lists and hence will inherit improvements.[10]

The data from our experiments show that geobuckets offer significant speedups over lists and heaps, at a reasonably modest cost in space. There are other alternatives — see Appendix B for some of these possibilities — but we believe that geobuckets will prove to be competitive and simpler to implement.

A Geobucket Variations

Most of the variations below can worsen the worst-case running time by a small constant factor. But in return, they may actually reduce space and time costs. Space usage is improved by performing more (partial) canonicalizations to allow more cancelation. The linear searches of extractions are sped up by skipping over empty buckets, by making more buckets empty, and by incrementalizing runs of extractions. We expect these to have only moderate success for Gröbner basis computations because additions are separated by extractions: if summands are small, then (almost) all buckets will be searched anyway.

Underflow. If a non-empty bucket $B[i]$ contains d^{i-1} or fewer terms, then this could result in adding a big summand f to a small partial sum $B[i]$. This does not violate condition (1), but one might think that it would be better to *underflow*, i.e. to move the contents of bucket $B[i]$ bucket down to a lower bucket whenever cancelation (addition) or extraction results in fewer than d^{i-1} terms. If the lower bucket is non-empty, it can cascade. The resulting cascades and underflows will probably all be below $B[i]$, which is empty, and therefore likely can hold overflows.[11] The analysis in Section 5 of canonicalization shows that this takes $O(\#(B[i]))$ time and thus can slow things down. Also, a term can now be added to more buckets than originally assumed, so $O(\#(B[i]))$ more credits are necessary in the analysis. These costs suggest that this is not an effective strategy, and we did observe a slow-down when experimenting with the 4-by-4 example.

[10] Improvements to our list polynomial code brought the time for 4-by-4 example down from 20 hours to 16.

[11] This is guaranteed for d sufficiently large; see the Remarks in Section 5.

Dynamic Capacities. It should be clear that different bucket capacities can result in different merging behaviors, so, depending on the structure of incoming summands, it may be worthwhile to adjust capacities. Doing so intelligently requires knowledge about the structure of summands and would probably benefit from using linked buckets (see below) to move buckets around more easily. Otherwise, our choice of d is still good, since it balances quadratic behavior in buckets with the frequency of cascades.

Anchored Addition. An initial addition has cost $\#f + \#(B[i])$, where $i = \lceil \log(\#f) \rceil$. One might initially be tempted to use a smaller i — say, $i = 1$ — to reduce this cost. Of course, $\#f$ violates the capacity of $B[i]$, so a cascade is likely,[12] requiring multiple inspections of the terms in f and offsetting the reduced cost of initial addition. Surprisingly, the worst-case analysis is unaffected: condition (1) still holds, terms still move up at most $\log N$ buckets, and the larger a polynomial $\#f$ is during a cascade, the less the cost per term. But the cost of multiple inspections means that it might be better simply to partially canonicalize before addition (see below).

Partial Canonicalization. A large summand f presents an opportunity to consolidate all lower buckets: canonicalize (force a cascade) on all lower buckets before performing the initial addition. That is, compute $g = B[1] + \cdots + B[i-1]$, $h = g + B[i]$, and $h + f$. As shown by the analysis of canonicalization in Section 5, the cost of computing h is bounded by the capacity of $B[i]$, which is $O(\#f)$. The cost of computing $h + f$ requires at most $\#g$ more comparisons than $f + B[i]$ since the only new terms are those from g, but $\#g$ is also $O(\#f)$. The asymptotic complexity is therefore unchanged, and in fact the worst-case analysis already covers part of the cost.

Periodic Canonicalization. Canonicalizing the entire geobucket allows maximal cancelation to take place, but it cannot be done too frequently or $O(N^2)$ behavior will return. To control canonicalization, maintain a balance (think bank balance) of credits. Credit the balance with the number of terms inspected during each list polynomial operation. When the balance is large enough (e.g. a constant fraction of the total number of terms currently in the geobucket), canonicalize the geobucket and debit the balance by the number of times terms were inspected. This keeps the cost of canonicalization proportional to the cost of all other operations, so the asymptotic time complexity is no worse.

Array Doubling.[13] The technique Hopgood [Hop68] proposed for dealing with hash table overflow is a general way to perform a sequence $r_1 < r_2 < \cdots < r_k = r$ of resize requests in $O(r)$ time and space:

```
Allocate an array $B$ of size $r = 1$;
for each $r_i$ do if $r_i > r$ then
    $r$ := max($2r$, $r_i$);
    Allocate $B'[1..r]$;
    Copy $B[1..r_{i-1}]$ into $B'[1..r_{i-1}]$;
    Use $B'$ as $B$
```

At each point, the array is at most twice as large as needed, so at most half the space is wasted and hence takes $O(r)$ space. Since new arrays are always at least double in size, at most $1 + 2 + 4 + \cdots + 2^j \simeq 2^{j+1}$ elements are copied, for some j such that $2^j \geq r > 2^{j-1}$. This overhead for copying is at most a factor $2^{j+1}/r < 2^{j+1}/2^{j-1} = 4$ extra over initializing an array of size r and hence takes $O(r)$ time. Since $r \leq \lceil \log N \rceil$ for geobuckets, this is only an additional $O(\log N)$ time and space, which is negligible.

Linked Buckets. In place of an array, we could use a linked list of non-empty buckets. Accessing the bucket for an initial addition would not be appreciably slower since the search for

[12] If cascades are suppressed, then $O(N^2)$ behavior can occur.

[13] Any constant factor greater than 1 will work, but 2 seems like a good space-time tradeoff.

the initial bucket is not much more costly than computing the logarithm. Buckets are created as they are needed, so this does not affect asymptotic running time, and it uses minimal space. This technique could speed up actual running time for extraction if partial and periodic canonicalization clear out many buckets.

Incremental Extraction. Bigger buckets are likely to contain more terms because of their larger capacity and therefore are likely to contribute more lead terms. Consider performing canonicalization by repeated extraction. At full capacity, bucket $B[i+1]$ is likely to contribute roughly a factor d more lead terms than $B[i]$. Analysis of canonicalization shows that if $B[m-i]$ is inspected with relative frequency $i+1$ to other buckets, then the total cost for extraction can be linear.

To achieve this behavior for a sequence of extractions, modify the code in Figure 2 to maintain the value j_i of j at each bucket $B[i]$. Thus $j_i \leq i$ points to the bucket $B[j_i]$ that contains the lead term of the first i buckets. Note that our method of performing cancelation on the lower bucket obviates the need to recompute j_i's. Given these pointers, the lead term is in bucket $B[j_m]$. These pointers can be maintained lazily with a low-water mark indicating the highest bucket with valid information. When a bucket is changed during addition or extraction, the low-water mark is moved, if necessary, to keep it below the change. Pointers are then computed as needed.

B Alternative Approaches

We assume that summands are list polynomials, since the summands in Gröbner basis computations are naturally obtained in sorted order. Recall that the list representation is simple, works equally badly online as offline, supports constant-time lead term extraction, and uses minimal space since cancelation is performed as soon as possible. The major disadvantage is thus its $O(N^2)$ worst-case running time, as exhibited in the 4-by-4 example.

Offline Techniques: Sorting and Priority Queues

Two simple techniques for offline addition are sorting and using priority queues. However, it is not immediately clear how to obtain online variants or how to deal with interspersed extractions, which add an online flavor. Incrementality is needed, but the obvious sorting approach of taking advantage of the sorted order of the partial sum and summands by merging is exactly that of list polynomials.

To use sorting, sort all the terms, canceling during sorting, afterwards, or at both times. There are at most N cancelations, so this has worst-case running time $O(N \log N)$. Unfortunately, most sorting algorithms fail to take advantage of the sorted order of monomials within each polynomial f_i, so they have an expected running time of $O(N \log N)$, i.e. a potential $O(\log N)$ factor slowdown over list polynomials.

Since adding small polynomials to large ones is bad, we can use a priority queue to always add the smallest polynomials together:

```
Place the summands into a priority queue
while the queue holds 2 or more polynomials do
    Remove the two smallest polynomials from the queue
    Insert their sum back into the queue
```

This uses and maintains the order of terms in summands. In the worst case, no cancelation occurs during addition, so the size of a sum is at least twice the size of the smaller summand.[14] A term

[14] The resulting tree of additions has the same shape as that obtained by Huffman encoding if size is interpreted as

can therefore be involved in only $\log N$ sums; in each sum, each term is examined at most once. Therefore, there are at most $O(N \log N)$ comparisons. If there is a lot of cancelation, the actual running time can be much better — as low as $O(N + n \log n)$, where $n \log n$ is the overhead of maintaining the priority queue.[15]

Heaps and Balanced Trees

The fast $O(\log N)$ or even $O(1)$ running time of *meld* (heap union) and $O(\log N)$ running time of *deletemin* (extraction of a minimal element) might make certain classes of heaps appear attractive, e.g. *binomial heaps* [Koz92, Bro78]. To perform the sum, the polynomials are converted into binomial heaps if necessary, using at most $O(N)$ time, and then melded in $O(n)$ time, which by assumption (Section 2) is $O(N)$. The lead term can be extracted by repeatedly applying *deletemin* and canceling like terms until reaching a term with a different monomial. To canonicalize, repeatedly extract the remaining lead term. This technique works both online and offline and has a worst-case running time of $O(N \log N)$. Binomial heaps therefore look like a good alternative.

However, binomial heaps suffer from the same problems as sorting, viz. the order within the f_i's is lost, the expected overall running time is $O(N \log N)$, and the expected space usage is $O(N)$: since cancelation is performed only when lead terms are extracted, most terms must be stored, so $O(N)$ terms are in a heap-ordered tree and must travel $O(\log N)$ places to reach the root.

This pessimistic estimate of space appears to be accurate: in one test run, approximately 50 times more space than necessary was used (see Table 4).[16] Also, we found that binomial heaps ran faster if, during lead term extraction, we more aggressively performed cancelation as follows. During the linear search of *deletemin*, keep track of which binomial trees contain the lead monomial. Afterwards, recursively rip apart binomial trees and search for any more occurrences of the lead monomial. Finally, perform cancelation on all terms containing the lead monomial and rebuild the binomial heap from the remaining trees.[17]

A slightly faster approach is to use binomial heaps of list polynomials instead of binomial heaps of terms. The order within summands is no longer lost, which helps a little because when the terms in the tail do not need to be permuted when the lead term is extracted. However, the tail as a whole must be repositioned to maintain heap-order, and the space usage is the same as before, since cancelation is still only performed upon extraction.[18]

Other heap structures are also problematic. Any heap-ordered structure, e.g. *leftist* heaps [Tar83], will suffer from similar problems of space and failure to use inherent order. One can also used some kind of ordered, balanced tree. These allow maximal cancelation and deal well with size differentials, but are more complicated to implement.

frequency: the number of monomial comparisons and the number of bits are exactly the same, viz. the product of the size and the distance to the root of the tree. The difference is that if cancelation occurs, geobuckets take advantage of it, whereas nothing analogous to cancelation can occur in Huffman encoding.

[15] Priority queues are attractive for aggregate offline multiplication because it is even more costly than addition.

[16] In 1992-1993, Pedersen and Scarlet considered using binomial heaps but decided that they were unlikely to perform better than geobuckets; one concern was that pointer overhead for binomial heap's would double space usage compared to geobuckets [PS96]. This constant factor overhead is clearly overshadowed by unperformed cancelation.

[17] Besides the lead term, other like terms might be discovered. It is probably worthwhile to perform cancelation in these cases, too. This can result in multiple rounds of cancelation and rebuilding, but should help to reduce space.

[18] Aggressive cancelation during extraction helps here, too, but if cancelation is performed whenever like terms are detected, then heaps of terms could perform better: terms in tails of list polynomials are not exposed to cancelation.

Hashing

Hash tables support roughly constant time access, so addition can be performed in $O(N)$ time simply by inserting terms and resizing the table as necessary to accommodate overflow. Space usage is good since cancelation is easily performed during insertion. To support lead term extraction, Zippel [Zip96] suggests maintaining a small prefix of lead terms, from which lead terms are extracted. To add a summand, merge it with the original prefix up to some point to obtain a new prefix, and hash the remaining terms of the prefix and summand; note that if the prefix is exhausted during merging, then the remaining terms of the summand cannot be included in the new prefix since their relation to hashed terms is unknown. Thus, the prefix shrinks or grows depending on how much of the summand precedes the last term of the original prefix, and how much cancelation occurs therein.

If the prefix becomes empty from extraction or cancelation, then it must be rebuilt by searching through the hash table. This is somewhat costly, requiring at least time linear in the contents of the table.[19] If prefixes are kept too short, then frequent rebuilds are likely, incurring $O(N^2)$ behavior; if prefixes are kept too long, e.g. all terms, then summands are likely to penetrate deeply into prefixes during merges, again incurring $O(N^2)$ time.

We do not yet have theoretical or empirical data to guide our choice of prefix length. Indeed, we are not sure there is any reasonable strategy, but perhaps something like one of the following would work. Maintain a weighted average size a of summands, e.g. $a := .9 \cdot a + .1 * \#f$, and keep the prefix within a constant factor of a; when rebuilding the prefix, make it as long as allowed by this bound. Or, maintain a balance. Each term in a summand contributes a number of credits. Each inspected, uncanceled term in the original prefix costs a credit. This allows occasional deep penetration by a small summand. When rebuilding the prefix, charge the cost to the balance.

Tries

Frequently, monomials can be encoded as vectors of integers such that the order on monomials is the same as the lexicographical order on vectors.[20] A polynomial can then be represented as a *trie* indexed by vector components.[21] When applicable, this technique supports asymptotically fast addition and lead term extraction. But it may be hard to get an efficient implementation (the constant factors may be large), and the implementation will be tedious, especially if space is to be used efficiently, due to non-uniform distribution of vector components.

C Polynomial Ideals used for Benchmarks

The field of coefficients for all benchmarks is the set of integers modulo 17.

The generators for the 4-by-4 example are the 16 entries (one of them is redundant) of $AB-BA$,

[19] Given a table with H elements, a prefix of $p \ll H$ terms can be computed in $O(H)$ time and space by first computing the p-th largest term α using a linear-time order statistic algorithm, extracting from the table all terms larger or equal to α, and then sorting the prefix. Or, the prefix can be computed in $O(p)$ space and $O(H \log p)$ time by maintaining a heap of the p largest elements while scanning the table, and then linearizing the heap.

[20] In the commutative case, any admissible order can be encoded using vectors of reals [CLO92]; integer components suffice for most standard orders. Macaulay uses this technique for fast monomial comparison [BS94].

[21] Our code and Macaulay use tries to represent sets of polynomials [BS94]; this significantly speeds up the search for summands during reduction.

where A and B are

$$A = \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \qquad B = \begin{bmatrix} q & r & s & t \\ u & v & w & x \\ y & z & A & B \\ C & D & E & F \end{bmatrix}.$$

The monomial order used is lexicographic on $F > \cdots > A > z > \cdots > a$. This is better than graded reverse lexicographic, but not as good as the orders used in [Hre94]. (We tried using the first order given in [Hre94] and found that using geobuckets gave speedups of roughly two or three times over lists.)

The generators for the cyclic roots of unity of degree d are $p_d = x[0] \cdots x[d-1] - x[d]^d$ and the following d polynomials:

$$p_k = \sum_{i=0}^{d-1} \prod_{j=0}^{k} x[(i+j) \bmod d], \qquad 0 \le k < d.$$

The monomial order used is graded reverse lexicographic on $x[0] > \cdots > x[d]$.

References

[Bro78] M. R. Brown. Implementation and analysis of binomial queue algorithms. *SIAM Journal of Computing*, 7:298–319, 1978.

[BS94] David Bayer and Mike Stillman. Macaulay 3.0: A system for computation in algebraic geometry and commutative algebra. Source and object code available for Unix and Macintosh computers. Contact the authors, or download from `math.harvard.edu` via anonymous ftp, 1982-1994.

[CLO92] David Cox, John Little, and Donal O'Shea. *Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra.* Springer-Verlag, New York, 1992.

[Hop68] F.R.A. Hopgood. A solution to the table overflow problem for hash tables. *Computer Bulletin*, 11(4):297–300, 1968.

[Hre94] Freyja Hreinsdóttir. A case where choosing a product order makes the calculations of a Groebner basis much faster. *Journal of Symbolic Computation*, 18:373–378, 1994.

[Koz92] Dexter C. Kozen. *The Design and Analysis of Algorithms.* Springer-Verlag, New York, 1992.

[Mis93] Bhubaneswar Mishra. *Algorithmic Algebra.* Springer-Verlag, New York, 1993.

[Mor94] Teo Mora. An introduction to commutative and non-commutative Gröbner bases. *Theoretical Computer Science*, 134:131–173, 1994.

[PS96] Paul C. Pedersen and Benjamin Scarlet. Private communications. July 1996.

[Tar83] Robert Endre Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, Pensylvania, 1983.

[Zip96] Richard Zippel. Private communication. June 1996.