Reduction of Huge, Sparse Matrices over Finite Fields Via Created Catastrophes

Carl Pomerance and J. W. Smith

CONTENTS

1. Introduction
2. Description of the Method
3. Outline of Experiments
4. Conclusion
Acknowledgements
References

The authors' research was supported in part by NSF grants.

We present a heuristic method for the reduction modulo 2 of a large, sparse bit matrix to a smaller, dense bit matrix that can then be solved by conventional means, such as Gaussian elimination. This method worked effectively for us in reducing matrices as large as 100,000 × 100,000 to much smaller, but denser square matrices.

1. INTRODUCTION

The last stage of several algorithms to factor integers, notably the quadratic sieve [Pomerance et al. 1988] and the number field sieve [Buhler et al.], involves the reduction of a huge, sparse bit matrix. (By a *bit matrix* we mean a matrix over the field with two elements: every element is 0 or 1 and arithmetic is modulo 2.) Although the matrices involved can be truly huge—100,000 rows and columns is no longer unusual—they may have as few as twenty 1's per row. Another feature of these matrices is that the first few columns tend to be denser than the rest: typically, about half of all of the 1's appear in the first $\sqrt{M}$ columns, out of a total of M columns.

In this paper we describe a heuristic method for obtaining linear dependencies among the rows of such matrices. In addition, we present data from experimental runs with randomly generated square matrices of size 50,000 and 100,000.

Odlyzko [1985] introduced a simple, two-step algorithm for dealing with these matrices. First, a certain fraction of the columns (the heaviest ones, which are mostly on the left side of the matrix) are declared "inactive", and are temporarily removed from the sparse data structure holding the matrix. What is left is a very sparse matrix with fewer columns than rows. Odlyzko then follows the *Markowitz rule* for row reduction in the active

1058-6458/92 $0.50 per page

part of the matrix. This is a local rule that at each stage chooses as pivot an entry that minimizes the Markowitz count $(r-2)(c-2)-2$, where r is the number of 1's in the active part of the row and c is the number of 1's in the column. The Markowitz count is the trivial upper bound for the net amount of fill-in that may occur when the given element is used as a pivot. When the sparse active part is finally eliminated, the history of the row operations performed is recalled, and repeated on the dense part. This results in a much smaller square matrix that can then be reduced via conventional Gaussian elimination.

Note that since arithmetic is modulo 2, there is never a problem with numerical stability. Thus the Markowitz rule can be used in its pure form.

Odlyzko observed experimentally that, if enough columns are put in the inactive portion of the matrix, the entire sparse active portion can be eliminated without the Markowitz count ever becoming positive. That is, there is no fill-in at all in the active part of the matrix. We call such an event a *catastrophe*. (The word "miracle" has been suggested to us as more appropriate. We use "catastrophe" in the sense of R. Thom's theory, where a small change can produce a large effect.)

In our initial experiments with Odlyzko's algorithm, we found that, for matrices as dense as the ones we actually were encountering with factorization algorithms, we had to pay too high a price to produce a catastrophe. Namely, we had to declare too many columns inactive, so that the resulting matrix, while smaller than the original, was uncomfortably large for conventional Gaussian elimination. In addition, there did not seem to be any reliable rule for deciding what was the optimal number of columns to declare inactive at the start of the process.

Our method is based on Odlyzko's two-step algorithm, but with the following changes. First, a much smaller number of columns are initially declared inactive. Second, we only do Gaussian elimination on the active part so long as no rows fill in. When this becomes impossible, we declare a few more columns inactive. This typically allows a few more elimination steps to be performed. We continue in this way, trying to coax a catastrophe into occurring. When it finally does, we find that many fewer columns have been declared inactive than in Odlyzko's original method.

Our experiments suggest that when the average number of 1's per row in the active part of the matrix reaches a certain critical level of about 3.3 ± 0.3, the catastrophe is about ready to occur. An unusual graph-theoretic model may be needed to explain this phenomenon (see Section 4). In particular, the development of the "giant connected component" in the evolution of a random graph [Bollobas 1985] may be relevant to the explanation.

The coordinate recurrence method from [Wiedemann 1986], which uses the Berlekamp–Massey algorithm for finding the minimum polynomial of a linear recurrence over a finite field, can also be used for this problem. The idea here is that if A is an $n\times n$ matrix and $\vec{x}$ and $\vec{y}$ are fixed n-vectors, the sequence $A^k\vec{x}\cdot\vec{y}$ for $k=0,1,\ldots$ is linear recurrent, easily computable if A is sparse, and its minimum polynomial is likely to be an important factor of the minimum polynomial of A. After a little calculating, we decided that while the coordinate recurrence method should eventually be the method of choice for sufficiently large problems, for our matrices it was perhaps five times slower.

In [LaMacchia and Odlyzko 1991], other methods for reducing sparse matrices modulo 2 (and, more generally, over a finite field) are discussed, including the conjugate gradient method, the Lanczos method and the coordinate recurrence method.

2. DESCRIPTION OF THE METHOD

Our goal is to find several, say ten, independent row dependencies in our given matrix. This may be accomplished via Gaussian elimination with either row or column operations. It is convenient to use row operations because, while the rows are uniformly sparse, some columns are fairly dense. We thus are able to make vertical cuts in the matrix to segregate the heaviest columns in which we delay row reduction. The row reduction proceeds with very simple Gaussian elimination steps on the sparsest portion of the matrix. We must remember the history of operations on this sparse portion, since it must be repeated on the inactive portion later.

We attempt to find the row dependencies by row reduction until the matrix is in upper triangular form, and then use back-substitution. We are not concerned with permuting rows and columns so that the upper triangular form is actually visible;

this can be kept track of internally. After an element is used as pivot, we consider its row and column to have been *eliminated*, though of course a record of what was done must be kept to obtain the final dependencies.

By the *weight* of a column we mean the number of 1's in the column corresponding to rows that have not yet been eliminated. By the *weight* of a row we mean the number of 1's in the row corresponding to *active* columns that have not yet been eliminated. Thus, whenever more columns are declared inactive, row weights either remain constant or decrease.

Step 0. Identify a certain number of columns as "inactive". This, in effect, temporarily removes these columns from the matrix. We initially designate as inactive the 5% heaviest columns of the entire matrix. (For particularly sparse matrices, it may be appropriate to designate fewer columns inactive, but there is no particular reason with our method to ever initially designate more than 5% inactive.) Processing on the inactive columns is deferred; Steps 1–4 apply only to the active columns of the matrix.

Step 1. Eject columns of weight 0, since they play no role in any row dependency.

Step 2. Eliminate each column of weight 1 and the corresponding row, since such a row cannot be involved in any row dependency. This procedure may create more columns of weight 0 or 1, so we cycle through Steps 1 and 2 until there are no more columns of weight 0 or 1.

Step 3. Eject excess rows. Because of Step 1, we may now discover that our matrix is more overdetermined than it needs to be—the difference between the number of rows and columns may be higher than the number of row dependencies that we need. We just delete these rows from the matrix until the row surplus equals the number of target row dependencies (typically about ten in factoring applications). The choice of rows to be deleted is completely up to us. We eliminate the heaviest rows, though other strategies, such as deleting rows that contain 1's in many columns of weight 2, may be worthwhile. We cycle through Steps 1–3 until no more deletions are possible.

Step 4. Use rows of weight 1 to eliminate the corresponding column. This amounts to deleting the row and column from the sparse structure, recording what happened so it can be duplicated later in the inactive portion of the matrix. Repeat Step 4 until no more moves are possible.

If the original matrix is sparse enough, it is possible that at this point everything has been removed from our sparse data structure, leaving only the inactive columns to deal with. Repeating the history of row operations performed on the sparse part on the relatively few inactive columns then creates a small, near square (and dense) matrix that may be now reduced with conventional Gaussian elimination.

However, what do we do next if, at the end of Step 4, there is still a large matrix? Odlyzko [1985] originally proposed to proceed with Gaussian elimination on the sparse part in a manner so as to minimize fill-in. For example, we might next use a row that intersects a column of weight 2 in order to eliminate that column. After there are no more rows or columns of weight ≤ 2, we could look for rows of weight 3 intersecting columns of weight 3, and so on.

Our method is both simpler and apparently more effective. We simply return to Step 0, designating an additional 0.1% (rather than 5%) of the remaining columns as inactive. Again, we choose for this role the heaviest columns remaining.

This, then, is the entire method. We iterate Steps 0–4 until there is nothing left in the sparse part. On the second and subsequent passes through Step 0, we remove from the sparse structure the heaviest 0.1% of the columns present (actually the smallest integer not less than 0.1% of the number of columns).

The process must terminate because, if we apply Step 0 often enough, we remove everything from the sparse structure. In the worst case, if the input is a dense bit matrix, it is likely there will never be anything to do in Steps 1–4. We will continually repeat Step 0, calling columns inactive until nothing is left.

We consider the method successful if, at the conclusion, the inactive columns are few enough to be handled by conventional matrix methods. For example, with an initial square matrix having 64,000 columns, we may be left at the end with 10,000 inactive columns. This is few enough to handle conventionally.

For sparse inputs, the reduction winds down in an interesting way, which we did not at first expect. At some point in the reduction procedure the density and structure of the sparse portion of the matrix are such that Step 4 "explodes": when the column intersecting a row of weight 1 is eliminated, other rows acquire weight 1, and the procedure continues to propagate in this way until nothing is left. This is the catastrophe referred to in the Introduction.

Note that our algorithm is quite greedy. At no step do we increase the number of 1's in the sparse portion. We refuse to allow any fill-in at all. In addition, no row or column in the sparse portion ever gets heavier.

One variation of this scheme seems to work a little better in practice. Notice that it is not necessarily harmful to create heavier columns, since they may be deleted in the next pass through Step 0 anyway. We thus delay our return to Step 0, by following Step 4 with the following.

Step 5. Use each row of weight 2 to eliminate the lighter of the two corresponding columns. The row and column are deleted from the sparse matrix structure. The other column intersecting the deleted row is replaced by its sum with the removed column (since we are working modulo 2, the sum is the same as the exclusive-or).

The effect on the matrix would of course be exactly the same if we used the row to eliminate the heavier of the two columns, but the history record (the list of rows to which our weight-2 row is added) would be longer.

This step usually reduces the number of 1's in the sparse matrix by two (coming from the deleted row itself), but sometimes other 1's are deleted in this process. We now cycle through Steps 4–5 until no further reductions are possible.

Try as we might, we could not otherwise improve significantly on the simple scheme described above. For example, we tried changing the percentage of columns declared inactive at each run through Step 0 (5% and 0.1% for the initial and subsequent passes). For nearby choices, the results were similar, and for grossly different choices, the results were worse. We tried other schemes for declaring columns inactive, such as choosing columns that intersect many rows of weight 3. We tried allowing elimination to continue so long as the Markowitz count stayed nonpositive. The different variations we tried were often not worse than the above method, but we could not find any that were appreciably better, and the elegant simplicity of the above method eventually led us to stick with it, rather than a more complicated variant.

(We have heard from Odlyzko that, if the original matrix is very overdetermined, it pays to do Step 3 in stages, rather than all at once. If there are enough excess rows to eliminate, one may also allow steps that increase some row weights. In particular, Step 5 may be supplemented with a step that uses a row that intersects a column of weight 2 in order to eliminate that column.)

We note a few more technical points. After the second pass through Step 0, there is little or, more likely, nothing to do in Steps 1–3: we are essentially just cycling through Steps 0 and 4 (or Steps 0, 4 and 5). But occasionally there is some action in these steps, and it does not cost much to look. Also, the catastrophe, when it comes, comes in Step 3. Thus, we leave these steps in the main loop.

When a column is called inactive it may not be the same as in the original matrix: it may have been affected by earlier row operations, if we perform Step 5. The matrix of inactive columns on which we later perform the history operations must consist of original columns only. Thus, when we call a column inactive, we just note its number and compile the numbered columns from the original matrix at the end of the run.

3. OUTLINE OF EXPERIMENTS

We ran a program that implements the algorithm described in the previous section on 44 test matrices of various sizes and densities. More precisely, the program incorporates the algorithm until the occurrence of a catastrophe and the elimination of the sparse part of the matrix. Actually finding nontrivial row dependencies requires duplicating the recorded row operations on the inactive portion of the matrix, reducing the resulting dense matrix, and back-substituting. These operations require considerably more time than the sparse-matrix processing of Steps 0–5 above. We did not include them in our timings because they are of a routine nature and, except for the back-substitution step, are done with the normal $\{0,1\}$-encoding for dense bit matrices.

Our test matrices were randomly generated and designed to approximate the matrices that occur in factorization algorithms. In particular, for an $M \times M$ matrix occurring in factorization algorithms, the number of 1's in column i is approximately DM/i, for some constant D. This is not a theorem, but a rough observation supported by heuristics.

For each value of D ranging from 2 to 3 in increments of 0.1, we randomly constructed three square matrices of size $M = 50{,}000$, in such a way that the probability of each entry being 1 was D/i for $i > 2D$ and $\frac{1}{2}$ for $i \leq 2D$, where i is the entry's column number. We ran the program for each matrix on a Sun 3/160 workstation with 20MB of memory, and averaged together the benchmarks for each group of matrices having the same value of D. In addition, we ran the same experiment for $M = 100{,}000$ and the same values of D, using only one sample per value of D. The results are shown in Table 1.

	$M = 50{,}000$			$M = 100{,}000$		
D	C	W	T	C	W	T
2.0	3168	3.19	0:43	6476	3.25	1:05
2.1	3652	3.27	0:55	7296	3.37	1:44
2.2	4152	3.43	1:03	8446	3.33	2:08
2.3	4716	3.38	1:18	9339	3.29	2:38
2.4	5255	3.37	1:33	10380	3.49	2:54
2.5	5833	3.57	1:40	11485	3.45	3:33
2.6	6466	3.54	1:55	12732	3.58	3:43
2.7	7028	3.61	2:05	13964	3.60	4:06
2.8	7655	3.44	2:33	15211	3.63	4:32
2.9	8221	3.36	2:50	16510	3.65	5:30
3.0	8825	3.51	3:00	17566	3.68	5:46

TABLE 1. Benchmarks for the algorithm of Section 2, applied to matrices of size M and density $\min(\frac{1}{2}, D/i)$, where i is the column number. C is the number of columns that had to be made inactive before a catastrophe was triggered. W is the average number of 1's per row in the active sparse structure just prior to the catastrophe. T is the running time, in hours and minutes, on a Sun 3/160 workstation with 20MB of memory.

The number of 1's per row just prior to the catastrophe (denoted W) ranged from 3.01, for a matrix with $D = 2.1$ and $M = 50{,}000$, to 3.68, for the matrix with $D = 3.0$ and $M = 100{,}000$. As one can see from Table 1, there is a slight tendency for this number to increase as the density of the original matrix increases. Table 1 also suggests that there is an approximately linear relation between D and the number C of columns that are made inactive, at least for the given range of D values. Despite the imperfect relationship between W and the occurrence of the catastrophe, we could not find a better predictor with our data.

Further experiments, especially with smaller and larger values of D, would be of interest. It should be noted that, the larger D is, the more main memory is required for handling the sparse data structure. Of course, disk memory could also be used, but paging will slow down the process. It may also be of interest to do experiments with random models other than the D/i model described above. For applications to discrete logarithm problems [LaMacchia and Odlyzko 1991], it would also be good to try experiments over finite fields with more than two elements.

4. CONCLUSION

We now propose a graph-theoretic interpretation of our results. Consider the graph where the vertices are the active columns in the matrix and where two columns are connected by an edge if there is a row of weight 2 whose 1's are in these columns. Suppose this graph is connected. If just one more column is declared inactive, the remainder of the matrix will be eliminated using just Steps 1–4 in our algorithm. That is, the catastrophe is ready to occur. If the catastrophe is not ready to occur, we declare more columns inactive, which has the effect of reducing the average row weight in the active portion, and thus increasing the number of edges in our graph (and reducing the number of vertices). Thus the graph is now more likely to be connected.

Actually our situation is somewhat more complicated. Some row of weight 3 may be "promoted" to a row of weight 2 while we are eliminating a row of weight 1. Thus a graph that does not at first glance look ready for a catastrophe may indeed be ready. A full graph-theoretic interpretation of a catastrophe, then, should include not only pairs of columns, but also triples, quadruples, etc. It should be a *hypergraph.* There may be lurking here a theory of evolution of random hypergraphs analogous to the well-known theory for graphs. For graphs, it is known that if there are $\frac{1}{2} + \varepsilon$ times as many edges as vertices, it is highly likely that the graph has a connected component comprising a positive proportion of the vertices. Our results suggest that, if the average row weight is a little over three, a catastrophe is highly likely to occur. This suggests a possible theorem on sparse random bit matrices.

ACKNOWLEDGEMENTS

We take this opportunity to thank several friends who helped with various aspects of our work, notably W. R. Alford, Renet Lovorn, Colette Pirie and Randy Tuler.

REFERENCES

[Bollobas 1985] B. Bollobas, *Random Graphs*, Academic Press, Orlando, FL, 1985.

[Buhler et al.] J. Buhler, H. W. Lenstra, Jr. and Carl Pomerance, "Factoring integers with the number field sieve" (preprint).

[LaMacchia and Odlyzko 1991] B. A. LaMacchia and A. M. Odlyzko, "Solving large sparse linear systems over finite fields", pp. 109–133, in *Advances in Cryptology: Crypto 90* (edited by A. Menzes and S. Vanstone), Lecture Notes in Computer Science **537**, Springer-Verlag, Berlin, 1991.

[Odlyzko 1985] A. M. Odlyzko, "Discrete logarithms in finite fields and their cryptographic significance", pp. 224–314, in *Advances in Cryptology: Proceedings of Eurocrypt 84* (edited by T. Beth, N. Cot and I. Ingemarsson), Lecture Notes in Computer Science **209**, Springer-Verlag, Berlin, 1985.

[Pomerance et al. 1988] C. Pomerance, J. W. Smith and R. Tuler, "A pipeline architecture for factoring large integers with the quadratic sieve algorithm", *SIAM J. Comput.* **17** (1988), 387–403.

[Wiedemann 1986] D. H. Wiedemann, "Solving sparse linear equations over finite fields", *IEEE Trans. Information Theory* **32** (1986), 54–62.

Carl Pomerance, Department of Mathematics, University of Georgia, Athens, GA 30602 (carl@joe.math.uga.edu)

J. W. Smith, Department of Computer Science, University of Georgia, Athens, GA 30602 (jws@pollux.cs.uga.edu)

Received August 27, 1991; revised August 7, 1992