University of Ljubljana
Institute of Mathematics, Physics and Mechanics
Department of Mathematics
Jadranska 19, 1000 Ljubljana, Slovenia

Preprint series, Vol. 37 (1999), 645

OPTIMAL REPRESENTATION OF SPARSE MATRICES

Andrej Brodnik Milena Kovač
Špela Malovrh

ISSN 1318-4865

March 17, 1999

Ljubljana, March 17, 1999

Optimal representation of sparse matrices

Andrej Brodnik
Institute of Mathematics Physics and Mechanics, Ljubljana, Slovenia
Luleå University, Sweden and *IskraSistemi*, Slovenia

Milena Kovač
University of Ljubljana, Biotechnical Faculty, Zootechnical Department, Slovenia

Špela Malovrh
University of Ljubljana, Biotechnical Faculty, Zootechnical Department, Slovenia

March 16, 1999

Abstract

This paper introduces a novel data structure used to store sparse matrices optimally – minimizing the space of the matrix representation and the time complexity of an access to the matrix element. The size of our data structure is close to the information theoretic minimum – it differs in the second order term – and permits constant access to the matrix elements and a constant amortized time with a high probability for the insertion of a new element in the matrix and a deletion of an existing element.

We tested our solution in a setup of mixed model equations that is used for genetic evaluation of breeding values and estimation of dispersion parameters. The coefficient matrices are large (over 1 mio. of equations) and sparse. The new algorithm reduced the number of accesses to the data structure (at most two accesses) and reduced the time for building the equation system, especially in theworst cases.

1 Introduction and Motivation

Goal of animal breeding is selection of animals as parents to produce "best" offspring. Since animals transmit the genetic material to their offspring, animal breeders want to "measure" so called breeding value of animals. But what really can be measured in the animals are their phenotype values.

Phenotype is combination of genetic and environmental components. Through measuring animal's own performance and/or performance of their relatives, collecting data on relationship between animals, animal breeders put together all pieces of information that can be useful in genetic evaluation. To extract genetic value from a phenotype value, animal breeders developed many sophisticated methods.

Dispersion parameters are a tool that tells how to extract genetic value – in animal breeding jargon – how to predict breeding value. Hence, estimation of dispersion parameters is one of the major tasks for animal breeders. Genetic (co)variance components and its proportions are required for predicting breeding values, designing breeding programs and observing realized genetic trends.

Obviously, it is very important to get as good estimates of dispersion parameters as possible, although this is not always an easy task. Due to desirable statistical properties, maximum likelihood (ML; [13]) and restricted maximum likelihood (REML; [19]) methods are almost exclusively used for variance component estimation from the selected data – they are consistent, asymptotically normal, and efficient.

Because it eliminates or at least reduces bias due to selection ([17]), REML appears to be the method of choice if data from breeding programs are used. Variance component estimation is based on Henderson's mixed model equations (MME; [14]). Dispersion parameters estimation with REML/ML is computationally demanding. Memory requirements and CPU time are determined by the size of equation system, number of estimated dispersion parameters, and convergence rate. Estimating procedure includes three elements: setting up the coefficient matrix (CM), solving the system of equations, and optimization. Setting up the CM is basically independent of the technique used to solve the system. Because REML and ML are iterative methods, the CM has to be built repeatedly in every iteration. This, in turn, means that the procedure for setting up CM has to be very time and space efficient.

In this paper we present such an efficient method for setting up CM. In the course of presentation, we first, in the next section, develop a novel data structure for a graph representation. Its size is almost as small as possible – it is, up to the additive term, the same as the information theoretic minimum. The data structure is, in the same section, extended to represent also sparse matrices efficiently. The section is followed by two sections on a practical implementation and on its measurements respectively. We conclude with some directions for the future work.

2 Theoretical Basis

The efficiency of the presented solution is contributed mostly to observing two important characteristics of the problem: the *rate* at which the data are changing; and the *bound* on the size of an individual entity we are dealing with. In general, any problem can be characterized as a dynamic, semi-dynamic or static and the choice of the data structure heavily depends on the characterization – e.g. the "static version" of a binary tree is a sorted array. The main problem we study is semi-dynamic: if the value of the element is changed to non-zero, it is never reversed back to 0.

Traditionally the speed of the algorithm is measured by the number of comparisons it makes (cf. [9]). However, such a measurement is not accurate enough for the contemporary computers. Therefore since early nineties, we started to use in the algorithm design ever more popular *cell probe model* ([11]) or its variations (e.g. Practical RAM, ERAM, etc.). In this model, we count the number of memory accesses instead of comparisons. The crucial property of the cell (register) is its size ([18]) – the number of bits it contains – that brings us to the second characteristic mentioned above. It is only since recently (cf. [2, 1, 3]) that scientists started to study its influence on the size and efficiency of the solution to various problems. They discovered, that the *finiteness of the universe* permits development of much more efficient algorithms then the classical ones (cf. [2, 1, 3]). The solutions presented in this paper use the same techniques.

2.1 Graph Representation

A graph $G(V, E)$ is defined as a set of vertices V and a set of connections (edges – also called relations) between these vertices E. In general, the relations can be of an arbitrary degree, but for the sake of discussion we restrict them to binary relations.

Given a graph $G(V, E)$ and vertices $u, v \in V$ any representation of the graph must be able to answer the query whether there is an edge (relation) between u and v – $(u, v) \in E$; i.e.:

$$E\texttt{.Member(}\ (u,\ v)\ \texttt{)} \rightarrow \{\texttt{true}, \texttt{false}\} \equiv \texttt{Boolean}\ . \tag{1}$$

Without loss of generality we can assume that vertices v are non-negative integers $0 \leq v < |V|$.

There are two well known graph representations – an adjacency matrix and a list of edges. The sizes of representations are $|V|^2$ bits and $\Omega(|E| \cdot \log |V|)$ bits respectively. The time complexity of the query (1) using adjacency matrix is $O(1)$ while for the list of edges it depends on the data structure used. Using *perfect hash table* ([10]) the query time is $O(1)$ and the space complexity $(1 + o(1)) \cdot (|E| \cdot 2 \lceil \lg |V| \rceil)$.

The query (1) is in fact a *membership* query: is the key (u, v) in the set $E \subseteq V \times V$, where $V \times V$ is a finite universe of size V^2. Brodnik and Munro in [4] show that for a finite universe there exists a data structure which stores a subset E succinctly and still permits constant time membership queries. The size of this data structure is $(1 + o(1)) \cdot \left\lceil \lg \binom{|E|}{|V|^2} \right\rceil$ bits, that is up to the additive term the same as the information theoretic necessary number of bits. Thus we have:

Theorem 1 *Given a graph $G(V, E)$, there exists an algorithm which answer the query (1) in $\Theta(1)$ time using $(1 + o(1)) \cdot \left\lceil \lg \binom{|E|}{|V|^2} \right\rceil$ bits of space.*

The data structure used by [4] employs six different substructures depending on the *relative sparseness* of the set. The data structure can be made dynamic keeping the same space and time bounds though with a high probability only ([3]).

2.2 Matrix Representation

The matrix is in fact a relation between the row i and the column j with a value other than just Boolean – for our purposes a real value:

$$M\texttt{.Value(}\ (i,\ j)\ \texttt{)} \rightarrow \texttt{float}\ . \tag{2}$$

Without loss of generality, we assume that the most common value in the matrix is 0. Since values of i and j are bounded by $0 \leq i < m_i$, $0 \leq j < m_j$ (m_i being the number of rows and m_j the number of columns) so is their combination $x(i, j)$

$$0 \leq x \equiv x(i, j) = j \cdot m_i + i < m_i \cdot m_j\ , \tag{3}$$

which we use as a key for a lookup into the *dictionary on the bounded universe* (cf. (2)).

The data structure used to store the dictionary (and, hence, the matrix) is a generalization from Theorem 1. However, there are few problems with a direct applicability of the data structure: first, it is not clear how to handle modestly sparse subsets – that is matrices with the number of non-zero (NZE) in the range $(m_i m_j)^{1-\epsilon} \leq \text{NZE} \leq \alpha m_i m_j$ for $\epsilon < 1$ and $\alpha \leq 1/2$. Next, for graphs with more than $V^2/2$ edges we could use their co-graphs, which does not work for matrices. Finally, the data structure was designed for a static problem and its dynamic behavior is obtained through the randomization and de-amortization.

Hence, for the practical purposes, we simplify the data structure slightly and reduce the number of sub-structures from six to two: we use a two-dimensional array and perfect hash table only. The perfect hash table implements dictionary of tuples $(x(i, j), a_{i,j})$ for $a_{i,j} \neq 0$ ($x(i, j)$. The table is a two level data structure with the second level used to resolve collisions from the first level. On both levels we use a hash function of the form

$$h(x) = (x \cdot k \bmod p) \bmod s \tag{4}$$

where x is the key, k randomly generated factor for a given hash table and p prime number larger then the size of the table s.

When we search for the element, we hash on the first level to get an address in the table where it is either an element or a reference to the second level (hash) sub-table containing the element. If we found the element we need to check whether it is the searched element and we are done. On the other hand, if we found the reference we hash for the second time. If we use hash functions of the form (4) both times, Fredman et al. in [10] proved the existence of such k-s that the second level hashing is collision free while the size of the structure remains small. Moreover, they proved that if k is chosen randomly every second k with a high probability has the required property. This fact is crucial to make the structure (semi-)dynamic (see [7]). Roughly, when we want to insert an element into the sub-table that is full, we enlarge its size to twice the number of elements in it. Then we keep choosing k at random and re-hashing the elements until no collision occurs in the sub-table. Because of the mentioned property of k, we will need to re-hash at most twice with a high probability.

The first level and all second level tables are stored in the same array consisting of elements shown in Figure 1. The `element` consists of the row, column and value, while `subtable`

```
typedef union {
  struct element {
    int   i, j;          /* j >= 0 */
    float aij;
  };
  struct subtable {
    int   start;
    int   value;          /* value < 0, size= abs(value) */
    long long k;
} tItem;
```

Figure 1: Type of the data items in the array.

stores reference into the array where the sub-table starts, flag and the size s of the sub-table, and k used in the hash function (cf. (4)).

Both, the two-dimensional array and the hash table permit constant access time, and occupy

$$m_i \cdot m_j \cdot |v| \qquad \text{and} \qquad (1 + o(1)) \cdot \text{NZE} \cdot (\lceil \lg m_i \rceil + \lceil \lg m_j \rceil + |v|) \tag{5}$$

bits respectively. Here $|v|$ is the number of bits necessary to represent an element of the matrix.

Let us put pieces together. The decision which data structure to use at which matrix sparsity depends on the number of non-zero elements NZE. From (5) we get that for

$$\text{NZE} < \frac{m_i \cdot m_j \cdot |v|}{2 \cdot (\lceil \lg m_i \rceil + \lceil \lg m_j \rceil + |v|)} \tag{6}$$

we store tuples and otherwise the array ((6) also formally defines a *sparse matrix*). Thus:

Theorem 2 *Given an $m_i \times m_j$ matrix of $|v|$-bit elements, there exists a data structure of size $\min\left(m_i \cdot m_j \cdot |v|\,,\ 2 \cdot \text{NZE} \cdot \lceil \lg m_i + \lg m_j + |v| \rceil\right)$ bits, that supports worst case constant time queries and constant time with a high probability insertions.*

We need to make two final remarks: first, from the theoretical point of view, we would achieve an asymptotically better space bound were we using all six sub-structures mentioned in

[4]. Second, the computation of the function (4) is expensive because it involves a multiplication and two divisions. However, Dietzfelbinger et al. proved that two of them can be replaced by binary shifts ([6]).

3 Implementation and Testing

The coefficient matrices as applied to animal breeding problems are always sparse as defined in (6). The proportion of non-zero elements (NZE) rarely exceeds 1%, very often it is in order of 0.1% – in other words, it is always bellow the bound in (6). Therefore, we always use perfect hashing to store them.

For the first implementation of the algorithm, (version 1) FORTRAN 77 was used, because the rest of the program is also written in this programing language ([16]). The array of elements defined in Figure 1 is transformed into three arrays `IA`, `JA` and `A` storage scheme as by Duff et al. ([8]), while the union is handled by the `EQUIVALENCE` statement. Because FORTRAN 77 does not support long integers (64 bits), library of procedures for multiple precision computations had to be implemented. After the first testings, it came out that multiple precision arithmetic is a bottleneck. Therefore, the subroutine for hashing was re-written into C (version 2). The other time-expensive part was a system random number generator. Hence, we implemented our own (version 3) according to Knuth ([15]).

Program version 1 was tested on data sets from Table 1 with the first level table of size $1.00 \cdot$ NZE, $0.75 \cdot$ NZE and $0.50 \cdot$ NZE elements long. Versions 2 and 3 (with subroutines written in C) were tested only with the first level tables of size $1.00 \cdot$ NZE. Memory size of twice NZE elements was expected to be enough for the data structure. Average number of accesses, size of the data structure and the time spent were measured on Sun workstation Ultra 2. Altogether, more than 1000 runs of program were done, since the algorithm belongs to the group of so called "randomized algorithms".

	GILTS	GOATS4	GOATS2	BOARS
$m_i = m_j$	34,449	24,579	12,291	256,709
bound from (6)	395,577,867	205,660,338	52,545,628	915,270,982
NZE	743,229	2,487,714	763,023	11,203,849
average number of accesses in the original algorithm (version 0)	31.2	21.1	10.7	128.1

Table 1: Testing data sets.

4 Results and Discussion

Figure 2 shows that the reduction of the first level table-size decreased also the size of the complete data structure for all data sets, in spite of the second level – the part with sub-tables – increase. Three element sub-tables occupied between 35 and 41% of the memory reserved for the data structure. More interestingly, there was practically no sub-tables larger then five elements. Therefore, we slightly simplified collision handling on the second level and used linear probing

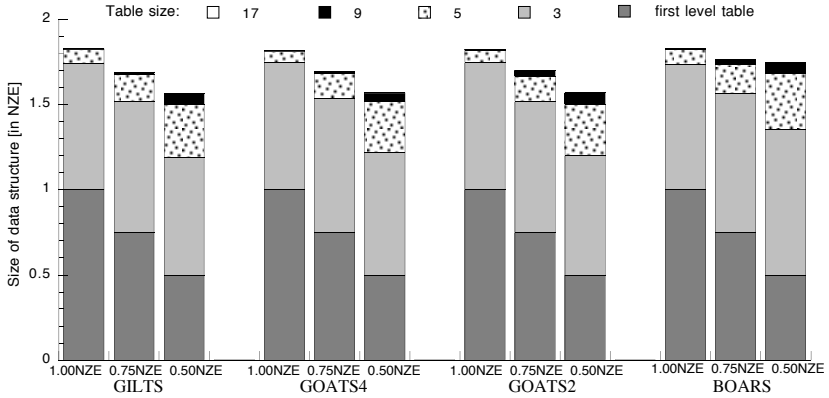

Figure 2: Size of data structure and distribution of sizes of sub-tables for the test sets.

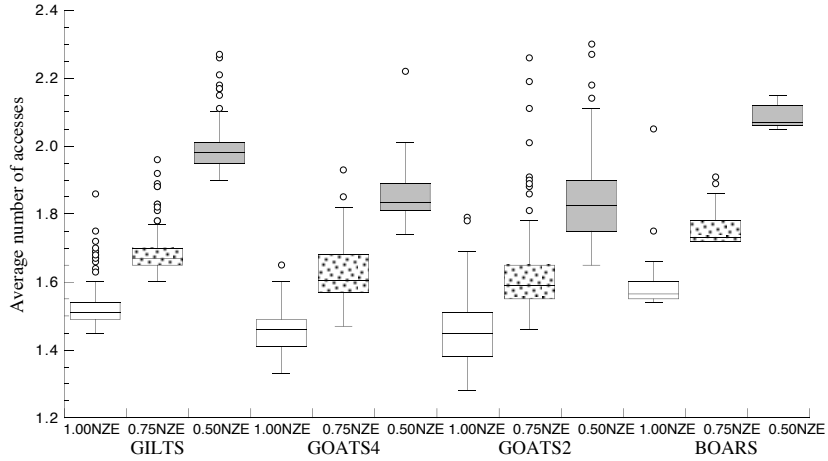

Figure 3: Average number of accesses for the test sets.

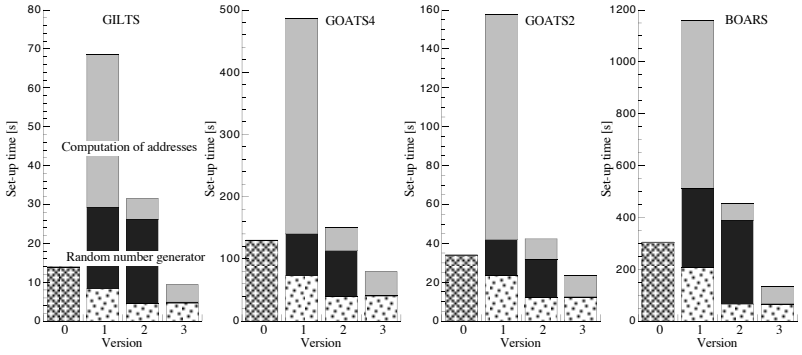

Figure 4: Set-up time components for the test sets.

(cf. [5]) instead of rehashing. Even using such a solution, the average number of accesses in the data structure rarely exceeded 2 as seen in Figure 3. With the set BOARS, average number of accesses was somewhat higher, while set GOATS2 had the most dispersed number of accesses. When we compare those results to the average number of accesses in the original version (31.2, 21.2, 10.7 and 128.1 for different test sets in Table 1), where a hashing with the linear probing was used ([12]), the improvement is obvious.

The time spent for setting up of MME in the original version was 4 to 5 times shorter than in the version 1 (FORTRAN 77 implementation) (Figure 4). In the version 1, up to 88% of total time (at the set GILTS 88%, 86% at sets GOATS4 and GOATS2 and 82% at the set BOARS) was spent for a multiple precision integer arithmetic and random number generation. The rest was comparable to the original version. Moreover, for a random number generation between 25 and 36% time was spent in the sets GILTS and BOARS and in the sets GOATS4 and GOATS2 approximately 15%. Differences in test sets are a consequence of a different number of accesses to a single coefficient in CM needed to compute its value. With switching from a multiple precision arithmetic in FORTRAN 77 to hashing subroutine in C, the time for the set-up of MME became comparable with the original algorithm. At that time, most expensive part of algorithm was generation of random numbers and with implementation of new pseudo-random number generator, we managed to cut time below set-up time of the original algorithm. Time for a generation of a random number in the last implementation was negligible for all test sets (Figure 3). The average time for a set-up in the last version was between 44% of the total time in the set BOARS and 69% in the set GOATS2. The original algorithm was quite successful for the set GOATS2 with only 10.7 accesses needed on the average (Figure 2). However, it was really slow for the set BOARS with 128.1 accesses. The difference in the cutting of a set-up time is a consequence of the difference in the number of accesses.

5 Future Work

For future work, we plan a number of improvements. First, we intend to decrease the size of the data structure by further reducing the size of the first level table. This will increase the size of the second level tables and hence (again) require hashing there. We will also re-implement the hash function to use its simplified version. Further, we intend to rewrite complete handling of the dictionary to deal with matrices with more than 2^{32} elements.

In the original version of the program, all memory is statically allocated which is due to a different handling of the sparse matrix. With our approach, we shall be able to allocate memory dynamically.

Our last version considers a matrix as a relation and hence stores its non-zero elements in a general-purpose dictionary. However, this raises an interesting question if observing the internal structure of the matrix would lead to further improvements.

At last but not least, we have to note that the setting-up of equations is just one of the tasks in the variance component estimating procedure. To improve the other tasks, we will have to employ parallelization, which, in turn, will require further changes in the setting-up as well.

References

[1] A. Andersson. Sublogarithmic searching without multiplications. In *36th IEEE Symposium on Foundations of Computer Science*, pages 655–663, 1995.

[2] A. Andersson, P.B. Miltersen, S. Riis, and M. Thorup. Static dictionaries on AC^0 RAMs: Query time $\Theta(\sqrt{\log n \log\log n})$ is necessary and sufficient. In *37th IEEE Symposium on Foundations of Computer Science*, pages 538–546, Burlington, Vermont, 1996.

[3] A. Brodnik. *Searching in Constant Time and Minimum Space (*MINIMÆ RES MAGNI MOMENTI SUNT*)*. PhD thesis, University of Waterloo, Waterloo, Ontario, Canada, 1995. (Available as technical report CS-95-41.).

[4] A. Brodnik and J.I. Munro. Membership in a constant time and minimum space. In *Proceedings 2nd European Symposium on Algorithms*, volume 855 of *Lecture Notes in Computer Science*, pages 72–81. Springer-Verlag, 1994.

[5] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, Massachusetts, 1990.

[6] M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen. A reliable randomized algorithm for the closest-pair problem. Technical Report 513, Fachbereich Informatik, Universität Dortmund, Dortmund, Germany, 1993.

[7] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R.E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, August 1994.

[8] I.S. Duff, A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices*. Monographs on numerical analysis. Clarendon Press, Oxford, 2nd edition, 1989.

[9] P. van Emde Boas. Machine models and simulations. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A: Algorithms and Complexity, chapter 1, pages 1 – 66. Elsevier, Amsterdam, Holland, 1990.

[10] M.L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, July 1984.

[11] M.L. Fredman and M.E. Saks. The cell probe complexity of dynamic data structures. In *21st ACM Symposium on Theory of Computing*, pages 345–354, Seattle, Washington, 1989.

[12] E. Groeneveld, M. Kovač, and T. Wang. PEST, a general purpose BLUP package for multivariate prediction and estimation. In *Proceedings 4th World Congress on Genetics Applied to Livestock Production*, pages 488–491, Edinburgh, UK, 1990.

[13] H.O. Hartley and J.N.K. Rao. Maximum likelihood estimation for mixed analysis of variance model. *Biometrics*, 54:93–108, 1967.

[14] C.R. Henderson. Estimation of variance and covariance components. *Biometrics*, 9:226–252, 1953.

[15] D.E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 2. Addison-Wesley, Reading, Massachusetts, 1969.

[16] M. Kovač. *Derivative Free Methods in Covariance Components Estimation*. PhD thesis, University of Illinois, Urbana-Champaign, Illinois, USA, 1992.

[17] K. Meyer and R. Thompson. Bias in variance and covariance component estimators due to selection on a correlated trait. *Journal of Animal Breeding and Genetics*, 101:33–50, 1984.

[18] P.B. Miltersen. The bit probe complexity measure revisited. In *Proceedings 10th Symposium on Theoretical Aspects of Computer Science*, volume 665 of *Lecture Notes in Computer Science*, pages 662–671. Springer-Verlag, 1993.

[19] H.D. Patterson and R. Thompson. Recovery of inter-block information when block sizes are unequal. *Biometrics*, 58:545–554, 1971.