GENBLOCK

Block Diagonalization of
VERY Large Binary Matrices
using the
Genetic Algorithm

Hugo de Garis

CADEPS Artificial Intelligence and Artificial Life Research Unit,
University Libre de Bruxelles, Ave F.D. Roosevelt 50,
C.P. 194/7, B-1050, Brussels, Belgium, Europe.
tel: +32 2 650 2783, fax: +32 2 650 2715
email: CADEPS@BRRNSF11(BITNET)

&

Center for Artificial Intelligence, George Mason University,
4400 University Drive, Fairfax, Virginia, VA 22030, USA.
tel: +1 703 764 6328, fax: +1 703 323 2630
email: HUGODEG@AIC.GMU.EDU

Keywords:
Block Diagonalization of (Very Large) Binary Matrices, Genetic Algorithm,
Operations Research, Optimization, Data Analysis, Sparse Matrices.

Abstract:
This paper shows that the Genetic Algorithm [Goldberg 1989] can be
profitably applied to the problem of block diagonalization of (very large) binary matrices.

Introduction:

Many problems in operations research can be transformed mathematically
into the problem of block diagonalization of binary matrices [Marcotorchino 1987], i.e.
how to permute the rows and columns of a binary matrix and then how to cut the rows and
columns into an equal number of sections such that the diagonal blocks contain as many 1s
as possible and the off-diagonal blocks contain as many 0s as possible. The quality of the
block diagonalization for a binary matrix is defined here as :-

\[
\frac{\text{#1s (inside diagonal blocks)} + \text{#0s (outside diagonal blocks)}}{\text{#rows} \times \text{#columns}}
\]

Broadly speaking, this paper consists of two parts. The first part introduces the
way in which the Genetic Algorithm can be employed to block diagonalize binary matrices,
and the second part presents an algorithm for an experiment to block diagonalize binary
matrices which are as large as possible, using only a work station. More specifically,
section 1 of this paper gives a brief explanation of the Genetic Algorithm [Holland 1975,
Goldberg 1989]. Section 2 provides a description of the Genetic Algorithm used for block diagonalization of binary matrices as well as descriptions of two experiments, one actually performed on "small" matrices (with the order of 1000 elements) and a second which is proposed for block diagonalization of very large binary matrices (with millions of (mostly zero) elements). Section 3 presents results of the first experiment. Section 4 proposes a "Sparse Matrix Algorithm" for the second experiment, and section 5 offers some conclusions and ideas for further research.

1. The Genetic Algorithm:

The Genetic Algorithm is a form of simulated evolution to solve optimization problems in a Darwinian "survival of the fittest" approach. Solutions to problems are coded onto (usually binary) strings called "chromosomes", (e.g. parameter values in a control problem), which compete with each other to reproduce the next generation. A quality value for the encoded solution of each chromosome is determined, and the probability of reproduction of each chromosome into the next generation is proportional to this value. The number of chromosomes per generation remains fixed. Genetic operators are applied to these strings, such as mutation (bit flipping), crossover (cutting two chromosomes at the same position and swapping portions), inversion (inverting a section of a chromosome). Occasionally, the application of these operators to the offspring causes them to have higher quality values than their parents. Hence they will reproduce with higher probability, and squeeze out inferior chromosomes. Over time, the average quality of the population will increase. The GA can be seen as a form of hill climbing where there may be many hills in the configuration space. For an excellent introduction to the principles of Genetic Algorithms see [Goldberg 1989].

2. Genblock:

The author of this paper has used GAs in several neural network applications [de GARIS 1989, 1990] and felt that the binary matrix block diagonalization problem might be solvable with the GA approach. This paper presents the results of such an experiment, which shows that the Genetic Algorithm can be successfully applied to the problem of block diagonalization of binary matrices. The results obtained were as good as the best obtained already in the literature (using such techniques as heuristic linear programming [Marcotorchino 1987]). This extremely satisfying result encourages us to extend our efforts in future work to much larger matrices. We assume that the GA is less computationally expensive (in terms of memory) than the alternative block diagonalization techniques. To further test this hypothesis, the technique developed in this experiment will be adapted in a further experiment aimed at block diagonalization of very large binary matrices. The aim of this second experiment will be to see how large a binary matrix can be block diagonalized on a personal computer within a "reasonable" time (i.e. within the limits of human patience, e.g. an overnight run). We believe that such an enterprise, if successful, may rouse considerable industrial and commercial interest. For example, Marcotorchino's team at IBM in France, are working on the problem of grouping synonyms for an electronic dictionary. This problem boils down to block diagonalizing a sparse binary matrix of the order of 30,000*30,000, with some 160,000 non zero elements. A matrix element of 1 indicates that the "row word" is synonymous with the "column word". At the present time, using the relational analytic techniques of Marcotorchino, this huge matrix has to be cut into chunks of about 300*300. It would be nice if the matrix could be treated as a whole. The aim of the second experiment will be to get as close as possible to this desirable goal.
2a. The Experiment:

The basic idea as to how to apply the GA to the block diagonalization of binary matrices is very simple. The user is asked to specify the number of cuts $C$ (implying that the number of diagonal blocks $D = C + 1$). A random row or column of the matrix is then moved randomly from one section to another. The quality (defined above) is measured after each move, either of a row or column.

The "chromosomes" of the population in this experiment are, in effect, a concatenation of the (permuted) numbers which identify the initial rows and columns respectively. In the experiment, each member of the population was associated with the following 2 data structures :-

a) The matrix
b) Parameter list

It was felt that the following 2 mutation types would be appropriate :-

i) Row transfer between 2 sections (with probability 50%)
ii) Column transfer between 2 sections (with probability 50%)

The choice as to whether a row or a column would be moved from one section to another was made with a probability of 50%. Two different sections were chosen randomly. A list of the row or column numbers (i.e. the numerical identifiers) is kept for each row or column section, plus the number of rows or columns in each section. Knowing the section from which a row or column is to be removed (the "from" section) and the section into which it is to be placed (the "to" section), one randomly picks a row or column from the "from" section and places it above or to the left of all the other rows or columns in the "to" section. The positions of the cuts are then updated appropriately.

A parameter list was kept for each matrix.

These parameters were :-

i) Row permutation
ii) Column permutation
iii) Rowcuts position list
iv) Colcuts position list
v) Row section counts
vi) Column section counts
vii) Row section rows list
viii) Column section columns list
ix) Number of cuts

Although this experiment is conceptually simple, it was not a trivial matter to program because one needed to calculate the "from" and "to" positions in the matrix before moving a row or column from one section to another. However the effort was worth it, because the results as shown in the next section were as good as the current best in the literature.

To initialise the population of chromosomes, the permutations of the rows and columns of the matrix were set to be in numerical order.
The quality measure

\[ Q = 2 \times 1s \text{ (in diag blocks)} + RC \times 1s \text{ (total)} - B \]

was converted to a computationally more convenient form (after a little algebra)

\[ Q = \frac{2 \times 1s \text{ (in diag blocks)} + RC - 1s \text{ (total)} - B}{RC} \]

where :-

1s (in diag blocks) is the number of 1s in the diagonal blocks
RC is the number of elements in the matrix
1s (total) is the number of 1s in the matrix
B is the number of elements in the diagonal blocks

In programming terms, a generation consisted of a single loop through the following sequence.

Initialization

gen: Quality measurement
    Scaling
    Reproduction
    Mutation
    Elite

In order to avoid premature convergence of a population of chromosomes to a sub "optimal" maximum value, (i.e. early "super" chromosomes reproduce too quickly in the early generations because their qualities are much higher than their rivals, yet may be far from "optimum") and to force the selection of superior fitness chromosomes when the average fitness approaches the maximum fitness in the later generations, quality measures can be linearly scaled using a scaling factor so that the modified scores do not differ from each other too much (nor get too close). These scaled scores are used for reproduction. See [Goldberg 1989]. Reproduction was performed using the roulette wheel technique, described as follows. The sum of the (scaled) qualities of the population was calculated, and the probability of a chromosome reproducing itself in the next generation was equal to its quality divided by this sum. However, before this reproduction occurred, the best chromosome (and its 2 data structures) was saved for insertion in the next generation (an elitist policy).

2b. A Proposed Second Experiment :

A second experiment is proposed for future work which will use the same basic idea as the first (i.e. shifting a row or column from one section to another) but will differ in its implementation. To test the quality of the block diagonalization, a perfectly diagonalized binary matrix will be formed (with \( Q = 1 \)) and then "mashed", by applying a large number of random permutations of rows and columns. This "mashed" matrix will become the matrix to be block diagonalized. Knowing its potential perfect quality (\( Q = 1 \)) will provide both a goal to be reached (if at all possible), and a basis for comparison.

As mentioned earlier in this section, it would be nice if huge matrices of the order of 30,000*30,000 could be diagonalised as a unit, without having to chop them up into
smaller chunks (e.g. of 300*300). The principal goal of this proposed second experiment will be to see just how large a binary matrix can be block diagonalized. Limiting factors which will determine this size will be (predominantly) the memory capacity of the workstation we use (a MAC II with 8 Megabytes of RAM), and the level of human patience required for the execution. If a given matrix can be diagonalized on our machine, the time limit will be an "overnight run", i.e. about 16 hours. With 8 Megabytes of RAM and 16 hours, how large a binary matrix will we be able to block diagonalize? This is the $64,000 question we hope to answer next.

Since our pre-mashed (starter) matrix will be large (starting at a size of 1000*1000 and increasing by increments of 1000 rows and columns) and sparse (i.e. most of the elements in the matrix will be zero), it is thought prudent to use sparse matrix techniques, e.g. manipulating lists of nonzero matrix elements. The proportion of 1s to 0s in our starter matrix will be kept at about the same value as in the IBM synonym matrix (i.e. 160,000/900,000,000 or about 1 to 5000). The initial number of cuts will be set to a value thought to be larger than the "optimum" number.

One can calculate roughly the memory needed for a large matrix. As an example, take the IBM synonym matrix. Theoretically this is a huge matrix of nine hundred million elements, but nearly all of them are zero. Only about 200,000 are 1s. Hence roughly 200,000 "entries" are required per matrix in the population. A entry needs to list the "coordinates" (i.e. the row and column number) of the non zero element. Since there are 30,000 rows/columns, 15 bits are required to specify each row and column, i.e. 30 bits, which fit nicely into a 32 bit machine word. Moving a row or column from one section to another implies a corresponding update of entries. With a population of 20 matrices, 20*200,000 words = 16 Megabytes will be needed. This is more than our MAC II has, but not a lot more. If the time factor is not too prohibitive, then a matrix of size corresponding to about 4 Megabytes of memory might be possible (i.e. of about 10,000*10,000). So, memory size seems not to be the overriding problem. The time taken to apply the GA to such a matrix is difficult to predict, and needs to be determined empirically.

3. Results of the Experiment:

FIG. 1 shows a 59*26 binary matrix to be block diagonalized taken from [Marcotorchino 1987]. FIG. 2 shows the result Genblock produced in roughly 3 hours running on a MAC II machine under Light Speed C, using the following GA parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of chromosomes in population</td>
<td>20</td>
</tr>
<tr>
<td>Number of generations</td>
<td>4000</td>
</tr>
<tr>
<td>Scaling constant</td>
<td>1.5</td>
</tr>
</tbody>
</table>

FIG. 3 shows the results of the qualities measured as a function of the number of cuts C specified by the user. Note that the values seemed somewhat independent of the number of cuts, although admittedly, some sections (and blocks) were very small for larger values of C. The best value was 0.875 for 6 cuts, (shown in FIG. 2) and is virtually identical with Marcotorchino's result of 0.881.
FIG. 2

<table>
<thead>
<tr>
<th>Number of cuts</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quality</td>
<td>0.849</td>
<td>0.869</td>
<td>0.848</td>
<td>0.860</td>
<td>0.875</td>
<td>0.848</td>
<td>0.864</td>
</tr>
</tbody>
</table>
4. The Proposed "Sparse Matrix Algorithm":

FIG.s 4 and 5 show two binary matrices, where FIG. 5 is the result of shifting column 4 in FIG.4 to replace column 1. The old columns 1, 2 and 3 of FIG. 4 each move one column to the right. This type of operation was described in section 2a, where a row or column was shifted from a "from" section to a "to" section.

In a sparse matrix, the majority of the elements are zero, therefore it is useful to employ matrix handling techniques which deal only with the nonzero elements. The matrices of FIG.s 4 and 5 can be converted into a form which is easier to deal with, as shown in FIG. 6. The first and third lines of FIG. 6 are called "row lists" and the second and fourth lines are called "column lists". In a row list, the "i"th pair of brackets (called a "sublist") contains the column numbers of those nonzero elements in the "i"th row of the matrix. Similarly for the column list. Either the row or the column list can be used in what follows.

\[
R = ( (1 4) (0 2 3 5) (1 4) (1 4 5) (0 2 3 5) (1 3 4) )
\]

\[
C = ( (1 4) (0 2 3 5) (1 4) (1 4 5) (0 2 3 5) (1 3 4) )
\]

\[
R' = ( (1 2) (0 3 4 5) (1 2) (1 2 5) (0 3 4 5) (1 2 4) )
\]

\[
C' = ( (1 4) (0 2 3 5) (0 2 3 5) (1 4) (1 4 5) (1 3 4) )
\]

FIG. 6
Row list R' and column list C' are the result of the operation shown in FIG.s 4 and 5. This modification of lists gives a clue as to how a sparse matrix can be treated. For a column operation, e.g. to take the “i”th column and to put it at the “j”th column position, take the “i”th sublist of the column list and put it at the “j”th position in the column list and then shift the intermediate column sublists each one place to the left or right (as appropriate). This is easily done for the column list. For the row list, things are a bit more complicated. Take the concrete case shown in FIG.s 4, 5 and 6. The old columns 1,2 and 3 get shifted over to the right one place, i.e. whenever a 1, a 2, or a 3 appears in a row sublist, it should be incremented by 1. Columns 0 and 5 did not change, so 0s and 5s remain unchanged in the sublists. The old 4s become 1s. Generalising, if a column at position "i" moves to position "j", then in all row sublists, replace all "i"s by "j"s, and increment all numbers appropriately lying between i and j. Finally, put the resulting numbers in a sublist into numerical order.

Computationally speaking, handling such lists can be done fairly easily by using pointers and standard linked-list techniques. For example, one could keep two arrays of pointers, one array for the row lists and another for the column lists. Sublists could be put into a linked-list (where each link in the chain consists of a pointer to a sublist and a second pointer to the next link). Sublists themselves could also be linked-lists, which would facilitate the numerical ordering of the numbers in a sublist.

To calculate the quality measure of a matrix implies counting of the number of 1s in the diagonal blocks. For a block whose rows extend from R1 to R2 and whose columns extend from C1 to C2, one merely looks at the R1th to R2th row sublists to see if any of the numbers in them have values between C1 and C2. Every time this happens, increment the count of 1s in the block by 1. (A similar calculation could also be performed using the column sublists).

The above algorithm is quite simple and using pointer and linked-list techniques as suggested, ought to be fairly efficient in both memory demands and execution times. It is therefore likely, when the time comes to implement it, we will see quite large sparse matrices (of the order of 1000*1000 and larger) being diagonalised.

What has not been mentioned so far, is the fact that the row and column lists have to be prepared in the first place. In the synonymy case for example, this could be done by giving each word in the 30,000 vocabulary a numerical ID, and then for each word finding its sublist (i.e. the IDs of its synonyms). In practice this might create difficulties if only a subset of the vocabulary can be treated in the memory space and time available. Hopefully it is possible to choose a subset of a reasonable size, all of whose synonyms are also in the subset. This may be more easily said than done. Nevertheless, it is obvious that the above algorithm requires a preparation phase to create the row and column lists.

5. Conclusions and Ideas for Future Research :

Applying the GA to the problem of binary matrix block diagonalization is new and it was not obvious at the beginning of this research whether the results would be interesting. FIG 2 shows that the technique is highly successful, and just as successful as the best results obtained so far in the literature (i.e. using a heuristic linear programming technique of [Marcotorchino 1987]).

However, the Genblock technique may offer an interesting advantage over the linear programming technique, and that is that its computational complexity is probably of a lower order. If so, the technique can be used on matrices that are too big to be handled
adequately by (heuristic) integer programming. This is because the GA takes a small number of coded solutions and progressively improves the average population performance. Progress may be relatively slow, but at least adequate solutions will be found, whereas this may be impossible with alternative methods due to combinatorial explosion. A theoretical or at least empirical study of the computational requirements (memory, execution time) of the Genblock technique and others would be worthwhile.

It would be nice if the above algorithm could be modified so as to determine automatically the number of cuts. One idea might be to start with more cuts than is thought to be optimum and when no rows exist between two row cuts AND no columns exist between two column cuts, then one of those row cuts and one of those column cuts could be just thrown away and the cut count decremented by 1. In practice, one will probably be left with no row or column between two row or column cuts, but some columns or rows between two column or row cuts. Handling this kind of problem requires further research.

Finally, it will be interesting to see with the proposed second experiment just how large a (sparse) binary matrix can be block diagonalised in an overnight run on a modest work station.

Acknowledgements:

I wish to thank my research colleague Christine Decaestecker for some interesting discussions on GenBlock. It was she who originally suggested moving one row or column at a time out of one section and into another. In an earlier version of Genblock, I was swapping pairs of rows or columns (a more restrictive operation, which gave me inferior results). She also suggested using a perfectly diagonalised matrix which could be "mashed" and used as a test case.

References:
