Computational Techniques for the Automorphism Groups of Graphs

K. Balasubramanian

Department of Chemistry and Biochemistry, Arizona State University, Tempe, Arizona 85287-1604

Received October 12, 1993[⊗]

The algorithm developed by Razinger et al. for the automorphism group of a graph is significantly improved by eliminating the CPU-intensive n^3 matrix multiplication operations. The CPU times of the two algorithms are compared. The new algorithm based on memory manipulation reduces the required CPU time for some graphs by a factor of ~3350. For example, the new algorithm took 1 min, 29 s of CPU time for a graph containing 16 vertices, while the algorithm of Razinger et al. took 82 h, 54 min, 26 s for the same graph on an IBM RS6000/580 system.

INTRODUCTION

Computer perception of the symmetry of a molecule or the associated molecular graph is very important in numerous applications[1-20] ranging from spectral simulation to the enumeration of isomers. Computer programs which enumerate chemical structures often yield two-dimensional connection tables called adjacency matrices. For a realistic organic compound with a given structural formula, several such two-dimensional connection tables are generated. Any further analysis or application, such as ^{13}C NMR spectral simulation, enumeration of stereoisomers, optical isomers, etc., requires automated computational algorithms for the rapid machine perception of symmetries of the associated graphs.

The symmetry group of a graph is quite different from the normal point group of a molecule in that the former is based on the conectivity of the graph while the latter is based on the invariance of the molecule under proper or improper rotational operations. The symmetry group of a graph, more commonly known as the automorphism group of a graph, is a permutational group comprising permutations of the vertices of the graph which leave the adjacency matrix (connectivity) of the graph invariant.

Computer perception of graph automorphism symmetry can be a very CPU-intensive problem. In general any permutation of the vertices of the graph in question which leaves the adjacency matrix (connectivity of the graph) invariant belongs to the automorphism group. The process of the construction of the automorphism group thus requires checking if a permutation of the vertices of a graph leaves the adjacency matrix invariant or not. In general there are $n!$ permutations of vertices for a graph consisting of n vertices. The checking process involves two matrix multiplications[1] which comprise $2n^3$ multiplication operations. Thus the problem in general involves $2n^3n!$ multiplication operations. This amounts to 7 257 600 000 multiplication operations for a graph containing 10 vertices.

There are several computational procedures for constructing the automorphism vertex partitioning the graphs[6-10,12-15]. Different computational procedures were compared recently, and their relative merits were discussed.[1,15] Where as there are several techniques for the automorphism partitioning of the vertices of a graph, there are very few methods for the generation of the automorphism group. Randić's canonical labeling procedure[17-20] produces both the automorphic partitioning of the vertices and the permutations in the automorphism group. The technique is especially useful for a graph which contains single automorphic vertex partitioning. In this case one could find the number of equivalent canonical labelings by inspection of the graph. However, automated computational algorithms which have to be applied to several graphs simultaneously do not enjoy the advantage of human visual perceptions and pattern recognitions. Such algorithms have to work with only adjacency matrices as these are the ones readily available and stored in the computers. Razinger et al.[1] developed a technique for the generation of the automorphism group using the vertex partition information generated by computational techniques such as the extended Morgan algorithm, the principal eigenvector algorithm, etc. This reduces the computational intensity of the problem from $2n^3n!$ multiplications to a smaller number of multiplications for a graph which contains several automorphism vertex partition sets, although for graphs which contain a single equivalence class of vertices the required number of operations is still $2n^3n!$. In the present investigation we develop a computational technique and a code which eliminates the CPU-intensive matrix multiplications completely. This brings down the required CPU time dramatically in several cases. We coded the two techniques in Fortran77 and executed the codes on an IBM RS/6000-580 work station. The newly developed code took dramatically smaller (sometimes a factor of 3350) CPU times for several graphs, as shown here.

COMPUTATIONAL TECHNIQUES

First we define the automorphism group of a graph. The adjacency matrix of a graph $\mathbf{A}$ is defined as

$$A_{ij} = \begin{cases} 1 & \text{if the vertices } i \text{ and } j \text{ are connected} \\ 0 & \text{otherwise} \end{cases} \qquad (1)$$

A permutation of the vertices of a graph can be described by a permutation matrix $\mathbf{P}$ defined as

$$P_{ij} = \begin{cases} 1 & \text{if the vertex } i \text{ is permuted to } j \text{ by the permutation} \\ 0 & \text{otherwise} \end{cases} \qquad (2)$$

A given permutation is in the automorphism group if its permutation matrix $\mathbf{P}$ satisfies

$$\mathbf{A} = \mathbf{P}^{\mathrm{T}}\mathbf{A}\mathbf{P} \qquad (3)$$

For a graph which contains n vertices there are $n!$ permutations matrices. Thus a brute force rigorous computational technique would require $n!$ such checkings to verify if eq 3 is satisfied. As seen above, eq 3 involves two $n \times n$ matrix multiplications. Since each matrix multiplication requires n^3 multiplication

⊗ Abstract published in *Advance ACS Abstracts*, March 15, 1994.

operations, the total number of multiplication operations for exhaustive checking of eq 3 is

$$T(n) = 2n^3 n! \qquad (4)$$

This does not include the CPU times required for the generation of the permutation matrix **P**.

Razinger et al.[1] recognized the significant reduction in the CPU time and the total number of operations if the graph in question has several equivalence classes of vertex partitions. That is, two vertices of a graph can be said to be equivalent if they belong to the same orbit in the automorphism permutation of the vertices of the graph. Vertices can thus be partitioned according to their equivalence. All vertices in an equivalence class would have identical connectivity properties.

Suppose the vertices of a graph can be divided into equivalence classes such that the first set contains n_1 nuclei, the second set n_2 nuclei, and so on. There are graphs which contain only one equivalence class. Of course for such graphs all vertices are equivalent. In this case no simplication or reduction is possible in the total number of operations if one used eq 3, since no partitioning of the vertices is achieved. However if the vertices can be partitioned into m equivalence classes such that the ith equivalence class consists of n_i nuclei, some simplification is achieved. This is because the vertices in different equivalence classes cannot be permuted. All permutations in the automorphism group must have their orbits within the same vertex partition. This means that the automorphism group Γ must be a subgroup of $S_{n_1} \times S_{n_2} \times \ldots \times S_{n_m}$, where, as seen above, n_i is the number of elements in the ith equivalence class. In symbols,

$$\Gamma \subseteq S_{n_1} \times S_{n_2} \times \ldots \times S_{n_m} \qquad (5)$$

The number of times eq 3 needs to be checked reduces significantly as there are more equivalence classes and less number of vertices in each class. It can be seen that the number of multiplication operations for the exhaustive search of automophisms in this case is

$$T(n_1,n_2,\ldots,n_m) = 2n^3\, n_1!n_2!n_3!\ldots n_m! \qquad (6)$$

To illustrate consider a graph which contains 16 vertices and suppose that the vertices are partitioned into 3 equivalence classes, the first class containing 8 vertices, the second and third containing 4 vertices each. The total number of multiplication operations reduce from 1.714×10^{17} to 1.902×10^{11} as a result of vertex partitioning. Consequently, the algorithm of Razinger et al.[1] brings about a reduction of $\sim 10^6$ in CPU time. Yet as shown here this is not the most efficient procedure due to the sheer number of multiplications and matrix generations which need to be carried out.

In the procedure developed here we recognize that while eq 3 is ideal as a rigorous definition of the automorphic permutation, it is not as useful in the implementation of an algorithm which generates the automorphism group. The reason is that each implementation of eq 3 involves two matrix multiplications and thus $2n^3$ multiplication operations. This has to be performed $n_1!n_2!\ldots n_m!$ times and in the worst case $n!$ times where n is the number of vertices.

Significant reduction in CPU time is achieved if one recognized the effect of eq 3 on the matrix **A** without performing the matrix multiplication. In order to achieve this consider as an example a general 4 × 4 matrix in symbolic form, shown as follows.

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \qquad (7)$$

We consider the permutation (12) (34) in the orbit notation of four objects. This can also be represented as

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 1 & 4 & 3 \end{pmatrix}$$

where the first row stands for the objects in a natural order while the second row represents the elements to which the first row members get permuted. That is, 1 goes to 2, 2 goes to 1, 3 goes to 4, and 4 goes to 3. The permutation matrix **P** for the above permutation is constructed and shown as follows.

$$\mathbf{P}(12)(34) = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \qquad (8)$$

The transpose of the matrix is shown as follows.

$$\mathbf{P}^{\mathrm{T}}(12)(34) = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \qquad (9)$$

The result of $\mathbf{P}^{\mathrm{T}}\mathbf{AP}$ is shown as follows.

$$\mathbf{P}^{\mathrm{T}}\mathbf{AP} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \qquad (10)$$

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} a_{12} & a_{11} & a_{14} & a_{13} \\ a_{22} & a_{21} & a_{24} & a_{23} \\ a_{32} & a_{31} & a_{34} & a_{33} \\ a_{42} & a_{41} & a_{44} & a_{43} \end{bmatrix} = \begin{bmatrix} a_{22} & a_{21} & a_{24} & a_{23} \\ a_{12} & a_{11} & a_{14} & a_{13} \\ a_{42} & a_{41} & a_{44} & a_{43} \\ a_{32} & a_{31} & a_{34} & a_{33} \end{bmatrix} \qquad (11)$$

Upon careful examination of the resulting matrix, it is seen that the overall matrix multiplication corresponds to interchange of the second row with the first row followed by interchange of second column with the first column. Likewise the third row is interchanged with the fourth row combined with the interchange of the third and fourth columns. Note that the permutation corresponding to the permutation matrix is (12)(34).

We consider the permutation (1234) or in another notation

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \end{pmatrix}$$

The result of matrix multiplication for this permutation is shown as follows.

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} =$$

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}\begin{bmatrix} a_{14} & a_{11} & a_{12} & a_{13} \\ a_{24} & a_{21} & a_{22} & a_{23} \\ a_{34} & a_{31} & a_{32} & a_{33} \\ a_{44} & a_{41} & a_{42} & a_{43} \end{bmatrix} = \begin{bmatrix} a_{44} & a_{41} & a_{42} & a_{43} \\ a_{14} & a_{11} & a_{12} & a_{13} \\ a_{24} & a_{21} & a_{22} & a_{23} \\ a_{34} & a_{31} & a_{32} & a_{33} \end{bmatrix} \tag{12}$$

Comparison of the final matrix with the initial matrix reveals that the following permutations of rows and columns must be done to arrive at the final matrix from the initial matrix: Move the first column of the starting matrix to the fourth column, the second column to the third, the third column to the fourth, and the fourth column to the first. Now move the first row of the resulting matrix to the fourth, second to the third, third to the fourth, and finally the fourth to the first row. Thus the permutation (1234) resulted in the corresponding permutation of the columns followed by the rows of the starting matrix.

In general a permutation in the two-row notation will have the following form

$$\begin{bmatrix} 1 & 2 & 3 & \dots & i & \dots & n \\ p_1 & p_2 & p_3 & \dots & p_i & \dots & p_n \end{bmatrix}$$

That is, the above permutation is read as 1 goes to p_1, 2 goes to p_2 ..., i goes to p_i ..., and n goes to p_n. This permutation will result in the following exchanges of columns and rows of the matrix **A**. Move first column to the p_1 column, the second column to the p_2 column, ...the ith column to the p_ith column..., and the nth column to the p_nth column. Subsequently, move the first row of the resulting matrix to the p_1th row, the second row to the p_2nd row, ...the ith row to the p_ith row..., and the nth row to the p_nth row. The final matrix thus obtained is the result of multiplying $\mathbf{P}^{\mathrm{T}}\mathbf{AP}$ matrices. The advantage of the permutation over matrix multiplications is that it cuts down the CPU time dramatically.

The above algorithm can be coded without actually permuting the rows and columns by simply locating the (ij)th matrix element of the new matrix in terms of an old matrix element as dictated by the permutation. This obviates the expensive process of interchanging rows and columns. To facilitate this, we cast the algorithm described in Table 1. As seen from Table 1, the algorithm here is elegant and simple. It basically takes the second row of the permutation as the input. Instead of actually switching rows and columns, in nested do loops over indices i and j, it compares a_{ij} with the permuted matrix element $a_{p_ip_j}$. The moment it reaches inequality it exits both the do loops yielding the result that the permutation is not in the automorphism group. However if the two elements agree, then the algorithm scans through other matrix elements. Only after all the elements are equal it exists the do loops, yielding the result that the permutation is in the automorphism group.

For symmetric matrices such as the adjacency matrix of a graph, further simplification is achieved since only the upper or lower triangle of the adjacency matrix needs to be compared. This is demonstrated in Table 2. For graphs which have no loops, yet another computational reduction is feasible, since comparison of diagonal elements can be omitted.

The advantage of the newly developed algorithm is that it eliminates $2n^3$ matrix multiplication operations and the generation of $n \times n$ permutation matrices entirely. This dramatically reduces the required CPU time for numerous graphs, as we shall demonstrate in the next section.

Table 1. Automorphism Group Algorithm without Matrix Multiplication

```
Given p_i for i = 1, 2, ... n describing the permutation.
  For i = 1, n  DO
    For j = 1, n  DO
      if a_ij ≠ a_pi pj  go to *
      else continue
The given permutation is in the automorphism group
  exit.
*The given Permutation is not in the automorphism group
  exit.
```

Table 2. Automorphism Group Algorithm for Undirected Graphs Containing No Loops

```
Given p_i for i = 1, 2, ... n
  For i = 1, n - 1  DO
    For j = i + 1, n  DO
      if a_ij ≠ a_pi pj  go to *
      else continue
The given permutation is in the automorphism group
  exit.
* The given permutation is not in the automorphism group
  exit.
```

COMPUTATIONAL IMPLEMENTATION AND COMPARISON

The author implemented both the algorithm of Razinger et al.[1] and the current algorithm in Fortran77 on an IBM RS/6000-580 workstation. It is noted that previously the algorithm of Razinger et al.[1] was implemented on a Vax workstation 3700. However in the current study both algorithms are compared on the same workstation. The generation of the permutations in the $S_{n_1} \times S_{n_2} \times ... \times S_{n_m}$ group was accomplished through a series of nested do loops wherein the NEXPER subroutine outlined by Nijenhuis and Wilf[21] was called. The NEXPER called sequentially generates the next permutation from the array containing the current permutation. The NEXPER subroutine generates even and odd permutations alternately.

In our version of the code we have also implemented the principal eigenvector generation algorithm to come up with an initial set of equivalence classes. The relative merit of this algorithm has already been discussed in several previous publications.[1,15] The algorithm discriminates all vertices correctly except for isospectral plaints. The algorithm runs as follows. One obtains the next vector of length n ($\mathbf{X}_{k+1}$) from the previous vector as follows

$$\mathbf{X}_{k+1} = \mathbf{A}\mathbf{X}_k \tag{13}$$

$$\mathbf{X}_0 = \frac{1}{n^{1/2}}\begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} \tag{14}$$

The starting vector is a normalized vector containing unities. The process is iterated n times, where n is the number of vertices. If the components i and j in the vector $\mathbf{X}_n$ are equal, then the vertices i and j of the graph are considered equivalent. This procedure yields the equivalence classes of vertices except for the isospectral points. Our code also has an option of providing equivalence classes as inputs or from other programs which use other techniques for the generation of equivalence classes.

Once the equivalence classes are constructed, the permutations of the $S_{n_1} \times S_{n_2} \times ... \times S_{n_m}$ group are generated. Note that m is the number of equivalence classes. For each permutation thus generated a subroutine called AUTOCHK is invoked which checks if the given permutation is in the automorphism group or not. It is noted that, in the two programs we compared, the difference was only in this subroutine. In one case (Razinger et al.[1]) AUTOCHK used the matrix multiplication technique based on eq 3. In the other case AUTOCHK was based on the algorithm outlined in Table 2. This facilitated exact comparison of the performance of the two codes.

The codes developed here also had options to terminate execution after a given number of permutations were found in the automorphism group. At this point the code stops execution, giving the current permutation. This is a valuable CPU-time-saving option. Once a few permutations in the automorphism group are found, one could use the closure and inverse properties of the group. That is, for example, that $\{g_1, g_2, g_3, ..., g_n\}$ is set containing m permutations with which we stopped the execution of the code. Since the automorphism permutations form a group, if $g_i, g_j \in G$, then $g_i g_j \in G$ for ($i = 1,2, ..., m - 1, j = i + 1, ..., m$). This property can be used to generate all products and see if the product exists in the current set. If not, the set can be extended to encompass these newly generated elements. The next property that can be used efficiently is the inverse permutation. If $g_i^{-1} \in G$, then $g_i \in G$. This is used for $i = 1, ..., m$ to check if all the inverses are already in the set. If not, the set can be extended to include these. Finally the generator idea is used. This states that if $g_i \neq e \in G$, then $g_i^2, g_i^3, ..., g_i^k \neq e \in G$. For all $i = 1, 2, ..., m$, the powers of g_i are constructed until the identity is reached and the set is extended to encompass these new elements. Now the program can be restarted with the current permutation as the last generated permutation. Here the last generated permutation is the one with the largest distance from the identity permutation in the NEXPER algorithm which generated odd and even permutations alternately.

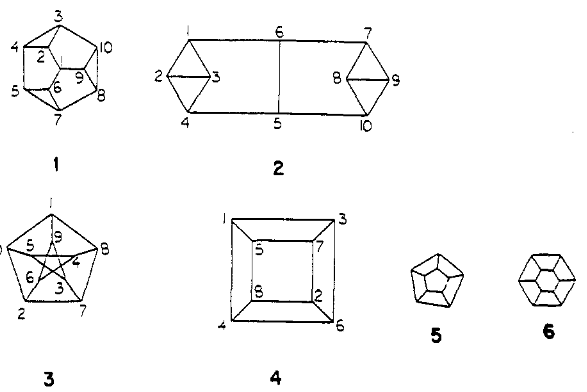

Figure 1. Six simple graphs for the CPU time comparison of the ...o codes considered here. See Table 3 for the relative performance.

Table 3. CPU TIme Comparison of the Two Algorithms

graph	CPU times[a] present	CPU times[a] matrix multiplication algorithm[b]	IIIa
L_{18}	2 s	5 s	2
1 (Figure 1)	<0.02 s	1.21 s	16
2 (Figure 1)	<0.03 s	4.52 s	6
3 (Figure 1, Peterson)	2 min, 10 s	3 h, 34 min, 56 s	120
4 (Figure 1, cube)	13 s	19 min, 49 s	48
5 (Figure 1, pentagon in pentagon)	16.6 s	1 h, 1 min, 43 s	20
6 (Figure 1, hexagon in hexagon)	39 min, 29 s	>137 h	24
anthracene	0.13 s	3 min, 28.7 s	4

[a] All CPU times were measured on an IBM RS/6000 580 workstation. [b] The algorithm from Razinger et al.[1]

Both versions, namely, that of Razinger et al. and the current version, were executed on an IBM RS 6000/580 workstation. The CPU times are now critically compared and discussed.

In order to compare the CPU times for the two codes first we consider a few simple graphs, shown in Figure 1. For linear unbranched chain graphs the CPU times taken were found to be quite small. Even for the L_{18} graph (chain graph containing 18 vertices) the CPU times taken by the two procedures are 2 and 5 s, as seen from Table 3. Of course it is well-known that the automorphism groups of all L_n chains contain just two elements. The first graph in Figure 1 took less than 0.02 s in the present algorithm, while it took 1.21 s using the Razinger et al. method. A similar situation is noted for the second graph in Figure 1.

Significant difference in the CPU times was found for the Peterson graph (third graph in Figure 1). The present algorithm took only 2 min and 10 s to generate all 120 permutations. However it took 3 h, 34 min, and 56 s of CPU time for the Razinger et al.[1] algorithm. The Peterson graph has only one equivalence class of vertices since all vertices are equivalent. Thus 10! permutations have to be generated and checked. The main difference is in the subroutine which checks if the generated permutation is in the automorphism group or not. In the Razinger et al. method $2n^3$ multiplications are made, while no multiplications are performed in the current

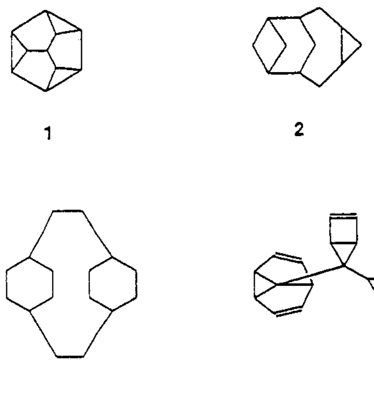

Figure 2. Four "complex" graphs from Razinger et al.[1] For the CPU time comparison, see Table 4.

procedure. This we believe is the key difference that dramatically reduces the CPU time for graphs with fewer equivalence classes. In this case the ratio of the CPU times was found to be 99.

The cube graph which contains eight vertices was considered as another test graph. The CPU time taken by the present code is 13 s compared to the CPU time of 19 min, 49 s for the Razinger et al. algorithm. In this case the ratio of the CPU times was found to be 91.

Next we consider graph 5 in Figure 1 (pentagon within a pentagon). This graph took considerably less time compared to the Peterson graph, although both graphs contain the same number of vertices and the same equivalence class structure. This is because there are 120 permutations in the automorphism group of the Paterson graph, while there are only 20 permutations for graph 5 in Figure 1. The code takes a long time to check if a permutation is a member of the group since it has to exhaustively search through the entire matrix to conclude if a permutation belongs to the group. On the other hand if a permutation does not belong to the group, then it can exit the do loops sooner. This explains why it took only 160.6 s for the graphs in Figure 1 compared to 2 min, 10 s for the Peterson graph using our code. Likewise the Razinger et al. method took 1 h, 1 min, 43 s for graph 5 (Figure 1), while the same method took 3 h, 34 min, 56 s for the Peterson graph. The ratio of the CPU times for the two codes for graph 5 (Figure 1) is 223.

The hexagon within a hexagon (graph 6, Figure 1) was found to be one of the most difficult test cases used in the current study. The CPU time taken by this graph was found to be 39 min, 29 s using our procedure, while the Razinger et al. algorithm took more than 139 h of CPU time and yet it had not completed the processing. In this case the ratio of the CPU times is estimated to be >208.

We also considered some polycyclic aromatic graphs. The anthracene structure took a mere 0.13 s using our code. The small CPU time is due to several equivalence classes. On the other hand triphenylene took 26 min, 40.4 s of CPU time using our code.

Razinger et al. have recognized a few graphs as "complex graphs" as they pose special problems for both the automorphism partitioning and the automorphism group. Figure 2 shows these graphs considered before by Razinger et al. We note that these authors used such graphs to illustrate the automorphism partitioning problem. The automorphism groups were not obtained before by Razinger et al. for these graphs. Table 4 compares the CPU times for the two techniques. Graphs 2 and 4 (Figure 2) should be considered

Table 4. Comparison of CPU Times for Some Complex Graphs

	CPU times[a]		
graph	present	matrix multiplication algorithm[b]	IVa
1 (Figure 2)	17.54 s	1 h 9 min, 21 s	6
2 (Figure 2)	0.06 s	12.97 s	4
3 (Figure 3)	1 m 29 s	82 h, 54 min, 26 s	16
4 (Figure 2)	0.03 s	0.81 s	8

[a] All CPU times were measured on an IBM RS/6000 580 workstation. [b] The algorithm from Razinger et al.[1]

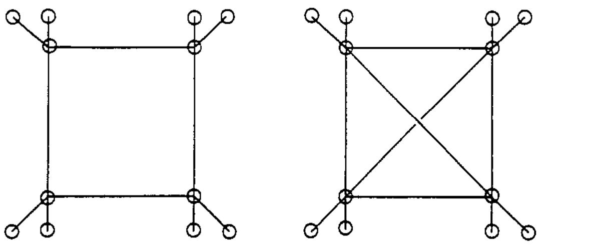

Figure 3. Few graphs whose automorphism groups are wreath products.

trivial from the stand point of CPU time. However graphs 1 and 3 (Figure 2) were more challenging. Graph 1 (Figure 2) took 17.5 s of CPU time using our method while the Razinger et al. technique used 1 h, 9 min, 21 s or a ratio of 237. Graph 3 (Figure 2) consumed 1 min, 29 s when the current algorithm was used, while the Razinger et al. algorithm took 82 h, 54 min, 26 s (a ratio of 3353). We also note that the principal eigenvector algorithm does not yield the correct automorphism partitioning for this graph. This graph, we believe, is the most dramatic comparison of the CPU times taken by the two algorithms.

Consider the left graph in Figure 3. Our current code took only 4.86 s to generate all 128 permutations in the automorphism group of this graph. It should be noted that the resulting group is the wreath product[22] of D_4 with S_2 denoted as $D_4[S_2]$. The wreath product groups and their extensions have been discussed in the literature before.[22] The automorphism group of the second graph (right) in Figure 3 containing 384 permutations took 5.95 s. The resulting group is the wreath product $S_4[S_2]$. This demonstrates the complexity of the generation of automorphism groups through exhaustive search methods.

It should be mentioned that our code can be enhanced to yield quick solutions for the complete and trivial graphs without going through automorphism checks as the automorphism groups of these graphs is S_n if n is the number of vertices.

We note that no single technique appears to be universally suited for all graphs. For example, for regular graphs which contain simply one equivalence class of vertices, the Randić canonical labeling method appears to be the best suited method. Often manual inspection of such graphs yields the automorphism groups. For example consider the cube graph. All eight vertices are equivalent. We label any vertex with the label 1. There are eight possible ways. Once this is done the three adjacent vertices to the vertex should carry the labels 6, 7, and 8 in the canonical labeling method. Since all three of these vertices are equivalent there are 3! ways of accomplishing this. Once this is done the rest of the labels are uniquely determined. Consequently the number of elements in the automorphism group is 8 × 3! or 48. Randić has applied this technique for several such graphs as the Desargues–Levi graph, the Peterson graph, etc., with considerable success.

As the author[11] showed the automorphism groups of all trees can be obtained by pruning the trees. The tree pruning algorithm yields the automorphism group of any tree readily as a recursive wreath or generalized wreath product. The algorithm is extremely efficient compared to all other algorithms described here.

Graphs which can be expressed as products of smaller graphs or compositions or generalized compositions of smaller graphs can be treated with more elegant and sophisticated techniques.[23] The automorphism groups of these graphs can also be expressed as generalized wreath products. It appears that the graph reduction algorithms could be superior to exhaustive search algorithms for the automorphism groups of such graphs. However, the universal applicability of the graph reduction algorithms remains to be tested. It appears at present that there cannot be a single technique that is uniformly elegant and efficient for all graphs. However a code can be developed which determines the special features of the graph in question and then decides which procedure is the most efficient one for that graph. For graphs containing several vertex partitionings such that there exist less than 12 vertices in each set, the algorithm developed here is certainly efficient.

REFERENCES AND NOTES

(1) Razinger, M.; Balasubramanian, K.; Munk, M. E. Graph Automorphism Perception Algorithms in Computer-Enhanced Structure Elucidation *J. Chem. Inf. Comput. Sci.* **1993**, *33*, 197–201.
(2) Balasubramanian, K. Applications of Combinatorics and Graph Theory to Spectroscopy and Quantum Chemistry. *Chem. Rev.* **1985**, *85*, 599–618.
(3) (a) Shelly, C. A.; Woodruff, H. B.; Snelling, C. R.; Munk, M. E. In *Computer-Assisted Structure Elucidation*; Smith, D. H., Ed.; American Chemical Society: Washington, DC, 1977; Vol. 54, Chapter 7, pp 92–107. (b) Gray, N. A. B. *Computer-Assisted Structure Elucidation*; Wiley: New York, 1986; Chapters 7 and 10.
(4) Christie, B. D.; Munk, M. E. The Role of Two-Dimensional Nuclear Magnetic Resonance in Computer-Enhanced Structure Elucidation. *J. Am. Chem. Soc.* **1991**, *113*, 3750–3757.
(5) Shelley, C. A.; Munk, M. E. Signal Number Prediction in Carbon-13 Nuclear Magnetic Resonance Spectrometry. *Anal. Chem.* **1978**, *50*, 1522–1527.
(6) Davis, M. I.; Ellzey, M. L., Jr. A Technique for Determining the Symmetry Properties of Molecular Graphs. *J. Comput. Chem.* **1983**, *4*, 267–275.
(7) Shelley, C. A.; Munk, M. E. Computer Perception of Topological Symmetry. *J. Chem. Inf. Comput. Sci.* **1977**, *17*, 110–113.
(8) Shelley, C. A.; Munk, M. E. An Approach to the Assignment of Canonical Tables and Topological Symmetry Perception. *J. Chem. Inf. Comput. Sci.* **1979**, *19*, 247–250.
(9) Herndon, W. C. Canonical Labelling and Linear Notation for Chemical Graphs. In *Chemical Applications of Topology and Graph Theory*; King, R. B., Ed.; Studies in Physical and Theoretical Chemistry; Elsevier: Amsterdam, 1983; Vol. 28, pp 231–242.
(10) Balaban, A. T.; Mekenyan, O.; Bonchev, D. Unique Description of Chemical Structures Based on Hierarchically Ordered Extended Connectivities (HOC Procedures). I. Algorithms for Finding Graph Orbits and Canonical Numbering of Atoms. *J. Comput. Chem.* **1985**, *6*, 538–551.
(11) Balasubramanian, K. Symmetry Groups of Chemical Graphs. *Int. J. Quantum Chem.* **1982**, *21*, 411–418.
(12) Morgan, H. L. Generation of Unique Machine Description for Chemical Structures. A Technique Developed at Chemical Abstracts Service. *J. Chem. Doc.* **1965**, *5*, 107–112.
(13) Razinger, M. Extended Connectivity in Chemical Graphs. *Theor. Chim. Acta* **1982**, *61*, 581–586.
(14) Wipke, W. T.; Dyott, T. M. Stereochemically Unique Naming Algorithm. *J. Am. Chem. Soc.* **1974**, *96*, 4834–4842.
(15) Liu, X.; Balasubramanian, K.; Munk, M. E. Computer-Assisted Graph Theoretical Construction of ^{13}C NMR Signal and Intensity Patterns. *J. Magn. Reson.* **1990**, *87*, 457–476.
(16) King, R. B. The Bonding Topology of Polyhedral Molecules. In *Chemical Applications of Topology and Graph Theory*; King, R. B., Ed.; Studies in Physical and Theoretical Chemistry; Elsevier: Amsterdam, 1983; Vol. 28, pp 108–122.
(17) Randić, M. On Discerning Symmetry Properties of Graphs. *Chem. Phys. Lett.* **1976**, *42*, 283–287.
(18) Randić, M. Graphs Representing Molecular Topology. *J. Chem. Phys.* **1974**, *60*, 3920–3928.
(19) Randić, M. Symmetry Properties of Graphs of Interest in Chemistry II. Desargues-Levi Graph. *Int. J. Quantum Chem.* **1979**, *15*, 663–682.
(20) Randić, M.; Davis, M. I. Symmetry Properties of Chemical Graphs VI. Isomerizations of Octahedral Complexes. *Int. J. Quantum Chem.* **1984**, *26*, 69–89.
(21) Nijenhuis, A.; Wilf, H. S. *Combinational Algorithms*; Academic Press: New York, 1975.
(22) Balasubramanian, K. The Symmetry Groups of Non-Rigid Molecules or Generalized Wreath Products. *J. Chem. Phys.* **1980**, *72*, 65.
(23) Balasubramanian, K. Group Theoretical Characterization of NMR Groups. *J. Chem. Phys.* **1980**, *73*, 3321.