MIT/LCS/TM-64

FINDING ISOMORPH CLASSES FOR COMBINATORIAL STRUCTURES

Randell B. Weiss

June 1975

TM-64

FINDING ISOMORPH CLASSES FOR COMBINATORIAL STRUCTURES

Randell Brent Weiss

Massachusetts Institute of Technology

May 1975

Dissertation submitted to the Department of Electrical Engineering and Computer Science on May 14, 1975 in partial fulfillment of the requirements for the Degree of Master of Science.

This research was supported by the National Science Foundation under research grant no. P4P0460-000.

ABSTRACT

A common problem in combinatorial analysis is finding isomorph classes of combinatorial objects. This process is sometimes known as isomorph rejection. In graph theory, it is used to count labelled and unlabelled graphs with certain properties. In chemistry, it is used to count the number of structures with the same chemical formula. In computer science it is used in counting arguments in proofs in complexity theory. In coding theory, it is used to partition sets of vectors into easy to handle sets.

This thesis presents three different algorithms for solving this type of problem and compares their timing and memory use. Some examples are given of how to apply the algorithms to graph theory and coding theory.

Thesis Supervisor: Vera S. Pless

Title: Research Associate, Department of Electrical Engineering and Computer Science

ACKNOWLEDGEMENTS

I would like to thank the following people, without whose help this work would never have attained its current state.

Vera Pless, for giving me so much encouragement and moral support to finish this work as well as being willing to read so many unfinished and unedited drafts.

David Grabel, for getting me interested in this research and for providing the initial groundwork for it.

Robert Cassels, for the many productive sessions we talked and argued over the work.

Robert Indik, for his keen eye in proof reading the drafts.

This research was supported by the National Science Foundation under research grant no. P4P0460-000.

A common type of problem in combinatorial analysis has been existence and enumeration of combinatorial structures, such as vectors, graphs, or designs. This is also encountered indirectly in chemistry, computer science, and communications theory. It often happens that the descriptions of such structures are not unique, and one must take care in the enumeration that no two descriptions correspond to the same structure. This process has sometimes been referred to as isomorph rejection.

With respect to graph theory, the first significant breakthrough in graph enumeration is due to Polya counting [4], yet this method cannot algorithmically produce representative graphs for each equivalence class under the isomorphism relation nor can it compute the number of labelled graphs to be found in each class. The algorithms described in this paper can do both of these things. For other approaches to this problem see [3], [4], [6], [10].

To find the equivalence classes for such structures, one can consider the action of a suitable group on them. Often one can also represent a combinatorial structure by a vector once an appropriate mapping has been found. Finding the appropriate mapping of a structure to a vector and the group that embodies the isomorphism you care about usually is not very difficult and will be discussed further below. To view the effect of a group on a vector it is convenient to represent the elements of the group as permutations of the components of the vectors and consider the action of a permutation on a vector. We define a vector orbit as the equivalence class of vectors formed by the

action of the group on a given vector. When the group is properly chosen, this is equivalent to the set of combinatorial structures isomorphic to a given structure. Conversely, given a group acting on a set of vectors, the group can be considered as defining the isomorphism between two structures found in the same vector orbit.

If we have a group of permutations on n points, we regard the points as component positions of vectors of length n. More precisely, let p(i) denote the image of the point i under the permutation p, and v(i) denote the i'th component of the vector v. Then the definition of the action of a permutation p on a vector v, denoted p*v, is $(\forall i \leq \text{degree};\ (p*v)(i) = v(p^{-1}(i)))$. Stated algorithmically, this would be:

```
procedure action(p, v1, v2, degree);
  Comment:  v2 = p*v1
  value degree; integer degree, i;
  integer array p, v1, v2;
  for i:=1 until degree   v2[ p[i] ]:= v1[i];
```

Suppose we have a set S of vectors of length n, and a permutation group G of degree n which is considered as acting upon the vectors in the manner described above. Further, suppose S is closed under G. Then two vectors are said to be in the same orbit if there exists some permutation in G that sends one into the other, i.e. $(\exists p \in G;\ p*v_1 = v_2)$. Note that this is an equivalence relation on S, since G is a group. Thus the action of G on S partitions S into equivalence classes, each class being called a vector orbit.

As a concrete example of what was described above we consider

undirected graphs with no self loops or multiple edges. Such a graph G can be represented by its adjacency matrix M which is a matrix of zeroes and ones defined as follows. If G has n vertices, M is an n by n matrix whose rows and columns are labelled by the vertices in G; there is a 1 in position (i, j) if vertex i is joined to vertex j and a 0 otherwise. Clearly M is symmetric, and all the information is given by the elements above the diagonal. By writing this latter out as a vector, m can be represented by a vector of zeroes and ones of length $(n^2-n)/2$.

Example: Suppose we want to find all non-isomorphic graphs with four vertices and two edges. We represent the adjacency matrix of such a graph by a vector of length six. For example, the graph 1-2-3 4 has adjacency matrix

$$\begin{matrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{matrix}$$

which can be represented by the vector (1, 0, 0, 1, 0, 0). Consider the action of S_4 on the vectors of length four and weight two (the weight of a binary vector is the number of 1's found in the vector). Each such action determines a permutation of the vertices of G. For example, the permutation (1 2) which sends the vector (1, 0, 0, 1, 0, 0) into (0, 1, 0, 1, 0, 0) reflects the isomorphism between the graph 1-2-3 4 and the graph 1-3-2 4.

Note that in this case the group S_4 must have degree 6 and that one cannot merely extend the generating permutations of degree 4 to degree 6. Instead we must find a new set of generating permutations

that when acting on a vector have the same effect as permuting rows and columns of the adjacency matrix. We know that the permutations (1, 2, 3, 4) and (1, 2) generate S_4 of degree 4. Consider the action of (1, 2, 3, 4) on the adjacency matrix; it sends (M[1, 2], M[1, 3], M[1, 4], M[2, 3], M[2, 4], M[3, 4]) to (M[2, 3], M[2, 4], M[1, 2], M[3, 4], M[1, 3], M[1, 4]). So the analog of (1, 2, 3, 4) in S_4 degree 6 is (1, 3, 6, 4) (2, 5). Similarly, (1, 2) sends (M[1, 2], M[1, 3], M[1, 4], M[2, 3], M[2, 4], M[3, 4]) to (M[1, 2], M[2, 3], M[2, 4], M[1, 3], M[1, 4], M[3, 4]), which corresponds to (2, 4) (3, 5). So the desired group is generated by (1, 3, 6, 4) (2, 5) and (2, 4) (3, 5).

We find that the vector orbit containing (1, 0, 0, 1, 0, 0) has twelve elements. Thus the graph has twelve isomorphic images according to the relabeling of the points. The total number of vectors of length six and weight two is 15 (if length = n and weight = w, then the number of vectors is C(n, w) where $C(n,w) = n(n-1)\cdots(n-i+1)/w!$), thus there is more than one unlabeled graph on four points with two edges. By partitioning the set of all vectors of length six and weight two into vector orbits according to the group S_4 of degree six, we find that there are two non-isomorphic graphs: 1-2-3 4 with twelve isomorphic labeled graphs and 1-2 3-4 with three isomorphic labeled graphs.

We have been using the problem of finding all non-isomorphic graphs on eight points with a fixed number of edges to test our algorithms. We represent each graph by a vector of length 28 determined by its adjacency matrix. If the graph has k edges, the vector has k ones and is said to be of weight k. Since we want all

non-isomorphic graphs with k edges, we want the orbit of graphs under the action of S_8. In particular we want a representative graph from each orbit and the number of graphs found in each orbit. We use S_8 because it has the effect of relabelling the graph through all possible labellings of the nodes in the graph. We translate this into a representation of S_8, of degree 28 and order 8! = 40,320, operating on the vectors described above. We have succeeded in finding all non-isomorphic graphs with seven edges or less. As there are a total number of 1,683,217 such graphs including isomorphic ones, it is clear why a computer is desirable as a tool for solving this problem.

Three algorithms to find vector orbits were developed. The first, VORB1, used a direct approach and considerable brute force. The second, VORB2, used a direct approach and many programming tricks. The third, VORB3 was a modification of VORB1 to solve a more general class of problems. All three of these have been implemented on the CAMAC (Combinatorial and Algebraic Machine Aided Computation) system, which is a unified collection of algorithms for manipulating groups and combinatorial objects [2], [8], [9].

Algorithm VORB1 finds the vector orbits by computing the entire orbit and storing these vectors in hash tables so that it can search for them each time to see if a vector has been found before. A hash table is a data structure designed for fast search.

Specifically, VORB1 starts with a group G given by generating permutations and a set S of vectors. The problem then is to determine the orbits of these vectors under G. The program starts by accepting

the first vector from S. Then all the generating permutations are applied to this vector, and all new vectors appearing, which we will call stage 1 vectors, are kept. Then all the generating permutations are applied to stage 1 vectors, and all new vectors appearing, called stage 2 vectors, are kept. This process continues in this fashion until no new vectors are found for the next stage. This completes the determination of the first vector orbit. The vectors are stored in a hash table for rapid comparisons. At this point the next vector of S is accepted and compared with the vectors in the first orbit to see if it is there. If not, then its orbit is generated and stored as above. If it is there, then the next vector in S is accepted. The whole procedure ends when all orbits of the vectors in S have been determined and stored. For each orbit, a representive vector and the number of vectors found in that orbit are printed out.

For this algorithm, the set S is the set of all vectors of a given length n and weight w. However, the set S could be any set that can be generated algorithmically or can be taken from a list. Definitions:

1. P = Set of generating permutations for group G.
2. n = Length of vectors to be acted upon this time.
3. w = Weight of vectors to be acted upon this time.
4. orbit.size = Number of vectors found in the orbit so far.
5. vector.generator(i,n,w) = Function which generates one at a time all vectors of a given length n and weight w. The value of vector.generator is the i'th vector in the set for $1 \le i \le C(n,w)$. The ordering of the vectors within the set is dependent upon the algorithm

and is of no consequence. This function determines the set S mentioned above. This procedure is implemented according to algorithm 368 from CACM by P. J. Chase.

6. vects = Hash table containing all vectors found so far, including those found in previous orbits. Implementation of the hash table is that described in [1].

7. queue = Queue of vectors found but not yet acted upon by the permutations in P. Note: This is not actually a separate data structure in the program. Instead it was implemented using vects and special pointers.

8. action = Procedure defined above that performs the action of a permutation on a vector.

Algorithm VORB1:

```
procedure VORB1(P, n, w);
```

Comment: Output a representative vector and the size of the orbit for each vector orbit found in the set of all vectors of length n and weight w acted upon by the group generated by the permutations found in P.

```
value n, w; integer n, w;
integer array P;
begin
integer i, orbit.size;
integer array p[1:n], v[1:n], v'[1:n];
hash table vects;
fifo queue queue;
vects := {};
for i=1 until C(n, w) do
   begin v := vector.generator(i, n, w);
   if not v∈vects then
     begin output(v);
     vects := vects ∪ {v};
     queue := {v};
     orbit.size := 1;
     while queue≠{} do
          begin v := take first vector from queue
```

```
          for all p∈P do
             begin   action(p, v, v', n);
             if not v'∈vects then
               begin  orbit.size := orbit.size + 1;
               vects := vects ∪ {v'};
               put v' into queue
               end
             end
          end
       output(orbit.size);
       end
    end
end
```

Algorithm VORB2 generates each entire orbit as VORB1 does but does not do any searching in hash tables and does not store all the vectors found. The set S of vectors for VORB2 consists of all vectors of a fixed length n and weight w. The basis for VORB2 is a bijection f mapping the vectors in S onto the natural numbers between 1 and C(n, w) inclusive. A bit vector B containing one bit for each element in S is set up. Initially all bits in B are zero. At this point, the first bit in B is zero, so the first vector chosen is $f^{-1}(1)$. The orbit of $f^{-1}(1)$ is computed by applying the generating permutations and maintaining stages of vectors as in VORB1 until the orbit is complete. When new vectors v are encountered, the f(v)'th bit in B is set equal to 1. After each vector has been acted upon by the generating permutations it is thrown away. The bit vector B contains all the necessary information concerning which vectors have been seen so far. When the orbit is complete, the first zero bit remaining in B, say the i'th bit, is determined and the next vector whose orbit is calculated is $f^{-1}(i)$. This process ends when the bit vector has no more zero bits. The program prints out the number of vectors in each orbit and a

representative for each orbit.

The particular function f is crucial to the working of this algorithm. With it one no longer needs to store all the vectors in a hash table nor generate all of the vectors in S. The function f is dependent on the set S and is used to represent S in the algorithm in a similar manner to the vector.generator function used in VORB1. It is defined as follows:

$$f(v) = 1 + \sum_{i=1}^{w} C(POSN(v,i)-1, i)$$

where POSN(v,i) = position in vector v counting from the right where the i'th 1 is found.

For example, consider the vector v = (1,0,1,0,1). In this case the length n = 5 and the weight w = 3. So by the definition above, f(v) = 1+C(0,1)+C(2,2)+C(4,3) = 1+0+1+4 = 6. For v = (0,0,1,1,1), f(v) = 1+C(0,1)+C(1,2)+C(2,3) = 1, and for v = (1,1,1,0,0), f(v) = 1+C(2,1)+ C(3,2)+C(4,3) = 10 = C(5,3).

To prove that f is a bijection, we will show that the size of the domain of f equals the size of the range of f and that f is one to one. The domain of f is the set of all vectors of length n and weight w; so the size of the domain is clearly C(n,w). To prove that the size of the range is C(n,w), we will show that the smallest element of the range is 1 and the largest element of the range is C(n,w), so f is a map into the set of integers between 1 and C(n,w). Finally we will show that f is one to one and thus also onto by showing that if vector v_1 is lexicographically less than vector v_2 then $f(v_1) < f(v_2)$.

Call the lexicographically least vector v_{min}, then $POSN(v_{min}, j) = j$,

$1 \le j \le w$. So $f(v_{min}) = 1+\sum_{j=1}^{w} C(POSN(v_{min}, j)-1, j) = 1+\sum_{j=1}^{w} C(j-1, j) = 1$. Call the lexicographically greatest vector v_{max}, then $POSN(v_{max}, j) = n-w+j$. So $f(v_{max}) = 1+\sum_{j=1}^{w} C(n-w-1+j, j) = C(n-w-1+w+1, w) = C(n, w)$ by using the identity $1+\sum_{j=1}^{w} C(r+j, j) = C(r+w+1, w)$.

Now we must prove that if $v_1 <_L v_2$ then $f(v_1) < f(v_2)$. Assume $v_1 <_L v_2$. Then there exists an i such that $POSN(v_1, i) < POSN(v_2, i)$. Define v' such that $\forall j \ge i$; $POSN(v', j) = POSN(v_1, j)$ and $\forall j < i$; $POSN(v', j) = POSN(v_1, i)-i+j$. Thus v' agrees with v_1 in all components from the position of the i'th 1 on up and has the first i 1's all in a block. For example, if $v_1 = (0, 1, 0, 1, 1)$ and $v_2 = (1, 0, 0, 1, 1)$, then $v' = (0, 1, 1, 1, 0)$. Thus $v_1 \le_L v' <_L v_2$. Since $\forall j$; $POSN(v_1, j) \le POSN(v', j)$, then $f(v_1) \le f(v')$. Now we will prove that $f(v') < f(v_2)$.

Define $r(j) = POSN(v', j)-1$ and $s(j) = POSN(v_2, j)-1$. By the definition of v', $\forall j > i$; $r(j) \le s(j)$.

So $1+\sum_{j=i+1}^{w} C(r(j), j) \le 1+\sum_{j=i+1}^{w} C(s(j), j)$. Now all we have to prove is that

$\sum_{j=1}^{i} C(r(j), j) < \sum_{j=1}^{i} C(s(j), j)$. By the definition of v', $\forall j \le i$; $r(j) = r(i)-i+j$. So $\sum_{j=1}^{i} C(r(j), j) = \sum_{j=1}^{i} C(r(i)-i+j, j) = C(r(i)-i+i+1, i)-1$. Now since $r(i) < s(i)$, $r(i)+1 \le s(i)$. Therefore $C(r(i)+1, i) \le C(s(i), i)$. So $\sum_{j=1}^{i} C(r(j), j) = C(r(i)+1, i)-1 < C(r(i)+1, i) \le C(s(i), i)$ and $C(s(i), i) \le \sum_{j=1}^{i} C(s(j), j)$. Therefore, $\sum_{j=1}^{i} C(r(j), j) < \sum_{j=1}^{i} C(s(j), j)$.

Therefore $f(v') < f(v_2)$. So $f(v_1) < f(v_2)$. Thus f is one to one and onto. Therefore, f is a bijection.

In the implementation of f, naturally it would be time consuming to compute and recompute the binomial coefficients needed to calculate f. So we first call a procedure that sets up a table of binomial

coefficients using the law of Pascal's triangle, C(i,j) = C(i-1,j)+ C(i-1,j-1), and then do table lookups whenever we need binomial coefficients to compute f. The procedure vsetup that follows creates the table of binomial coefficients, procedure vmap is the implementation of f, and procedure invmap is the implementation of f^{-1}.

```
procedure vsetup(n,w,table);

  Comment:  table[i,j]= C(i,j) where 0≤i≤n-1 and 0≤j≤w

value n,w; integer n,w;
integer array table;
begin
integer i,j;
for i:=0 until n-1 do
   for j:=0 until w do
      if j=0 then table[i,j]:= 1;
         else if i<j then table[i,j]:= 0;
         else if i=j then table[i,j]:= 1;
         else table[i,j]:= table[i-1,j]+table[i-1,j-1]
end

integer procedure vmap(v,n,table);

  Comment: vmap= f(v)

value n; integer n;
integer array v,table;
begin
integer i,j;
vmap:= 0;
j:= 1
for i:=0 until n-1 do
   if v[n-i]≠0 then
      begin
      vmap:= vmap+table[i,j];
      j:= j+1
      end
end

procedure invmap(k,v,n,w,table);

  Comment: v= f^-1(k)

value k,n,w; integer k,n,w;
```

```
integer array v, table;
begin
integer i, j, c;
j:= w;
for i:=1 until n do
   begin
   c:= table[n-i, j];
   if k<c or j=0 then v[i]:= 0;
      else begin
      v[i]:= 1;
      k:= k-c;
      j:= j-1
      end
   end
end
```

Definitions: same as in VORB1 and procedures above plus:

9. bit.vector = Bit vector of 0's and 1's of length C(n,w).

10. queue = same as in VORB1 but implemented according to Knuth vol. 1, p. 241.

Algorithm VORB2

```
procedure vorb2(P, n, w);

  Comment: Output representative vector and size of orbit for each
  vector orbit in the set of all vectors of length n and weight w
  acted upon by the group generated by the permutations in P.

integer array P;
value n, w; integer n, w;
begin
integer i, orbit.size, vnum;
integer array table[0:n-1, 0:w], bit.vector[1:C(n,w)], v[1:n],
                v'[1:n], p[1:n];
fifo queue queue;
initialize bit.vector to all 0's;
vsetup(n, w, table);
for i:=1 until C(n,w) do
   begin if bit.vector[i]=0 then
     begin bit.vector[i] := 1;
     orbit.size := 1;
     invmap(i, v, n, w, table);
     output(v);
     queue := (v);
     while queue≠() do
        begin v := take first vector from queue;
```

```
        for all p∈P do
          begin action(p, v, v', n);
          vnum := vmap(v', n, table);
          if bit.vector[vnum]≠1 then
            begin orbit.size := orbit.size + 1;
            bit.vector[vnum] := 1;
            put v' in queue
            end
          end
        end
      output(orbit.size)
      end
    end
end
```

Now we will compare the timing and memory use of algorithms VORB1 and VORB2. The dominant factor in the timing of both of these algorithms is the number of operations that must be performed on every vector in the set S. Since S has size C(n,w) and all of the operations on the vectors are proportional to n, the dominant term in the timing formula is $n \cdot C(n,w)$. Note that queue operations are simple pointer manipulations that are not proportional to n.

Let k be the number of generating permutations for the group. For VORB1, the operations with time proportional to n performed on every vector in S are: one vector.generator call, k+1 lookups in the hash table, k action calculations, and one store in hash table operation. For VORB2, the operations performed on every vector in S with time proportional to n are: k action calculations (which includes the store operation) and k vmap operations. Note that the vmap operation is approximately the same as a hash table lookup (except a little faster), thus the extra cost of VORB1 over VORB2 is the necessity for a vector generation operation, one more hash table lookup, and a store in the

hash table.

Considering this it is now simple to see conceptually the advantage of VORB2. This comes exclusively from the knowledge of the nature of S that allowed us to create the bijection f from S to natural numbers between 1 and C(n,w). This function f is actually the perfect hashing function that allows us to do away with storing the vectors in a hash table (with associated pointers to take care of collisions that result from a hashing function that maps many vectors to the same number). And since f is a bijection we then could use f^{-1} to take the place of the vector generator.

Note that by using the nature of the bit.vector (an encoded list of all vectors found so far and all vectors yet to be found) it is possible to avoid having to generate every vector. Instead we search through the bit.vector for a zero and the invmap of that point is guaranteed to be a vector not yet found in any vector orbit computed so far. The search through the bit.vector is very inexpensive since we can check an entire word at a time for a zero in some bit and since we never have to backtrack in our search (the entire computation of VORB2 will result in only one pass through the bit.vector from one end to the other). Thus the search is very efficient (see [5] for a related view of the search problem when computing orbits of permutation groups). Since finding a new vector in the bit.vector was cheap, we then could do away with generating every vector and need only generate as many vectors as there are vector orbits, which is a number usually very much smaller than the size of S.

In actual running time, VORB2 was approximately ten times faster than VORB1 as long as k was small (approx. two) and C(n,w) was large enough so that the set up time for the table of binomial coefficients and the bit.vector was negligible (in practice that turned out to be C(n,w) > 100). VORB2 has one other advantage over VORB1 besides the modest improvement in speed. VORB2 uses much less memory than VORB1, and when memory use is proportional to C(n,w) this is a very important factor.

In VORB1 it was necessary to eventually store every vector of S in a hash table. In the implementation we used, the hash table required 2.33 times as many bits as the number required to store all of the vectors. Since each vector has n bits and there are C(n,w) vectors in S, VORB1 required a little more than $2.33 \cdot n \cdot C(n,w)$ bits.

In VORB2, the bit.vector took the place of the hash table, and it required just C(n,w) bits. However, in VORB1 the queue did not have to be a separate data structure since it used the hash table and special pointers. In VORB2, the queue is still required but since the bit.vector cannot store the required information, a separate structure is necessary. The queue holds vectors found to be in the current vector orbit being computed which have not yet been acted upon by the generating permutations. For the worst case, suppose for a given group there is only one vector orbit in S, and suppose further that the number of generating permutations is large, then the size of the the queue would be a fraction less than the total number of vectors in S. So in the worst case the queue would require approximately $n \cdot C(n,w)$

bits, which would represent a very modest saving for VORB2 over VORB1.

However, note that the size of a vector orbit cannot be larger than the size of the group. If we are using vectors to represent an adjacency matrix of some kind, then given a group of size j! the number of vectors in S would be $C((j^2-j)/2,w)$. Thus for this kind of problem, which is quite common, the largest possible vector orbit is a small fraction of the size of S when C(n,w) grows large, in which case the size of the queue would also be a small fraction of C(n,w). Thus there are uses for VORB2 where the memory saving would be quite substantial. For the kind of problems we were using these algorithms to solve, VORB1 limited us to cases where the size of S was measured in the ten thousands, whereas VORB2 allowed us to solve cases where the size of S was over a million.

At this point one may think that VORB1 is clearly an inferior algorithm. Yet it has one saving grace that is of considerable importance. That is its flexibility to handle problems where S is quite arbitrary since S is defined in the algorithm by the vector.generator and it is easy to write a new vector.generator for a new class of sets S. Remember that VORB2 depends entirely on the existence of the function f which in turn depends entirely on the nature of S. In this case VORB2 is limited to solving problems where S is the set of all vectors of a given length and weight.

We will now rewrite VORB1 so that S will be the closure under G of a set of vectors given it by the user, one at a time (the closure is the union of all of the vector orbits under G of the given vectors).

This algorithm will be called VORB3. Define vector.input(v,n) to be a logical function whose value is true if the user types in a vector. The vector given by the user is returned in v (n is the length of the vector as before). To use VORB3, the user gives it vectors one at a time via the console until he is done. The algorithm will reply either with the size of the orbit or with a statement that the vector is already in another orbit.

Besides the straight forward application of this algorithm to find a single vector orbit at a time, it is also useful for the general case where one wants to count something that cannot be generated in any other fashion. The vector orbits then represent chunks of data which are easy to specify as isomorph classes, and the union of a number of such orbits is the entire set that one wanted to generate. As an example of this, algorithm VORB3 was used to solve two problems in coding theory where the set S in one case was the set of 888 vectors of weight 36 in the (36,18) symmetry code over GF(3), and in the other problem it was the set of 41,184 vectors of weight 60 in the (60,30) symmetry code over GF(3). In the first case there were two vector orbits and in the second case there were four vector orbits. These orbits were needed to compute the complete weight distributions of these sets of vectors [7].

Algorithm VORB3:

```
procedure VORB3(P,n);
```

Comment: For each vector typed in by the user at the console, find its vector orbit. If it is an orbit already found before, say that, else print out the size of the new orbit.

```
value n; integer n;
integer array P;
begin
integer i, orbit.size;
integer array p[1:n], v[1:n], v'[1:n];
hash table vects;
fifo queue queue;
vects := {};
while vector.input(v,n) do
   begin
   if v∈vects then output("vector found in previous orbit");
     else begin vects := vects ∪ {v};
     queue := {v};
     orbit.size := 1;
     while queue≠{} do
          begin v := take first vector from queue
          for all p∈P do
             begin   action(p,v,v',n);
             if not v'∈vects then
               begin  orbit.size := orbit.size + 1;
               vects := vects ∪ {v'};
               put v' into queue
               end
             end
          end
     output(orbit.size);
     end
   end
end
```

The difference between VORB1 and VORB3 is merely the substitution of vector.input for vector.generator, the replacement of the outermost do loop with a while loop, and the minor rearrangement of the first conditional statement to indicate the case where the input vector is already a member of a known orbit. Thus the timing and memory use is as bad as with VORB1, except that the size of S may be considerably smaller than C(n,w). Now it is more efficient to discard algorithm VORB1 in favor of either VORB2 if S is the set of all vectors of a given length and weight or VORB3 if S is the closure under G of the set of vectors given by the user.

SUMMARY

Finding isomorph classes is a common problem in combinatorial analysis and is also applied to problems in graph theory, chemistry, computer science, and coding theory. If we represent the combinatorial structures as vectors and consider the action of a suitable permutation group on these vectors, the vector orbits obtained can represent the isomorph classes. Three algorithms for calculating vector orbits, finding a representative vector for each orbit and finding the size of each orbit, were described. In addition, program outlines were presented for each algorithm.

After analysis of the timing and memory considerations for these algorithms we concluded that algorithm VORB2 was most efficient for solving problems where the set being partitioned into vector orbits is the set of all vectors of a given length and weight and that algorithm VORB3 was most efficient for solving problems where one wanted only selected vector orbits given a list of vectors. Because algorithm VORB3 stores all of the vectors in the orbit, it is also useful for handling chunks of data which cannot be generated in any other way.

REFERENCES

[1] J. Cannon, R. Gallagher, K. McAllister. "Stackhandler: A Language Extension for Low Level Set Processing, Programming and Implementation Manual". Technical Report No. 5. Department of Pure Mathematics. The University of Sydney. 1972.

[2] J. Cannon, J. Richardson. "The GROUP System for Investigating the Structure of Groups, User and Maintenance Manual". Technical Report No. 8. Department of Pure Mathematics. The University of Sydney. 1973.

[3] D. G. Corneil. "The Analysis of Graph Theoretical Algorithms". Technical Report No. 65. Department of Computer Science. University of Toronto. 1974.

[4] F. Harary, E. M. Palmer. Graphical Enumeration. Academic Press. New York. 1973.

[5] J. McKay, E. Regener. "Algorithm 482, Transitivity Sets". Communications of the ACM. Vol. 17, No. 8. August 1974. p. 470.

[6] D. M. Perlman. "Isomorph Rejection on Power Sets". SIAM J. Comput. Vol. 3, no. 3. September 1974. pp. 177-183.

[7] V. S. Pless, N. J. A. Sloane. "Self-dual Codes over GF(3)". To appear in Information and Control.

[8] V. S. Pless. "CAMAC". Proceedings of the Sixth Southeastern Conference on Combinatorics, Graph Theory, and Computing. Florida Atlantic University. Boca Raton, Florida. Feb 1975.

[9] R. B. Weiss. "CAMAC: Group Manipulation System". MAC Technical Memo 60. M. I. T. Cambridge, MA. 1975.

[10] R. B. Weiss, R. A. Cassels. "Vector Orbits". To appear as a Project MAC Technical Memorandum. M. I. T. 1975.