This article appeared in *Computational and Constructive Design Theory*, edited by W.D. Wallis, Kluwer Academic Publishers, 1996.

6

ON WRITING ISOMORPHISM PROGRAMS

William Kocay

Department of Computer Science,
University of Manitoba,
Winnipeg, Manitoba, Canada R3T 2N2
bkocay@cs.umanitoba.ca

ABSTRACT

This is a self-contained exposition on how to write isomorphism programs. It is intended for people who want to write isomorphism programs for combinatorial structures, such as graphs, designs, digraphs, posets, etc.

1 INTRODUCTION

This is an expository article aimed at graduate students or advanced undergraduates who want to write isomorphism programs, whether for graphs, designs, set systems, posets, or other combinatorial structures. The emphasis is on programming techniques. The program presented here is based on graphs, but the principles of the algorithms for other combinatorial structures are very similar. The most powerful general purpose graph isomorphism program currently available is without doubt B.D. McKay's C-language program *Nauty* [23][1]. See McKay [24] for a mathematical description of the algorithm used. The prototype of Nauty was the Fortran program GLABC, which formed a part of McKay's Ph.D. thesis [22]. Several other authors [10,19,21] have independently written general purpose graph isomorphism programs, including Kocay, whose *Groups & Graphs* [18][2] package contains a highly efficient Pascal graph isomorphism program.

This work was supported by an operating grant from the Natural Sciences and Engineering Research Council of Canada.

[1]Nauty is available via anonymous ftp from dcssoft.anu.edu.au, in /pub/nauty19.

[2]Groups & Graphs is available at ftp.cc.umanitoba.ca in the directory /pub/mac.

W. D. Wallis (ed.), Computational and Constructive Design Theory, 135–175.
© 1996 *Kluwer Academic Publishers. Printed in the Netherlands.*

This article appeared in *Computational and Constructive Design Theory*, edited by W.D. Wallis, Kluwer Academic Publishers, 1996.

These programs are different implementations of very similar algorithms. To the best of my knowledge, all existing general purpose graph isomorphism programs are based on the method of partition refinement. This is the most efficient method known to date. These algorithms are currently exponential in the worst case, but in practice are extremely efficient for most graphs. It may be possible to develop them into general polynomial algorithms. A polynomial algorithm for all graphs seems a very real possibility. One reason for this article is to make the algorithm and programming techniques more accessible to a wide audience. There is ample room for research into graph isomorphism algorithms. As will be seen, the programs are quite complicated and subtle. This is something of an obstacle, but not an insurmountable one.

In 1980, Furst, Hopcroft and Lux [12] discovered a sub-exponential algorithm for computing the automorphism group of a trivalent graph. It was later refined and improved by Galil, Hoffman, Lux, Schnorr, and Weber [14] into a $O(n^3 \log n)$ algorithm. This is a very efficient algorithm, based on finding block systems in a 2-group. It does not extend very well to graphs of higher valence (see Luks [20] and Hoffmann [16]). It seems to me that partition refinement offers more hope for a general polynomial-time algorithm. There are also a number of methods based on the linear algebra of the adjacency matrix (eg., see Bennett and Edwards [4]). These often work well for showing that two graphs are not isomorphic, but do not work for all graphs. In [15], Gismondi and Swart attempted to transform the graph isomorphism problem into a linear program and use the simplex method. The technique did not work, because it could not guarantee integral solutions, but uses some very interesting ideas.

There is a vast literature on graph isomorphism. We do not attempt to survey it here. For more information, see the bibliographies in [11] and [3]. A great many algorithms for graph isomorphism have been presented, mostly based on partition refinement. To the best of my knowledge, *Nauty* out-performs them all. This is an indication of the power of partition refinement and the importance of programming techniques. Nevertheless, even *Nauty* takes exponential time on certain difficult graphs. Development of a polynomial algorithm will require some new ideas and techniques. Corneil and Goldberg have presented a non-factorial algorthm in [9]. They use partition refinement with *sections* in order to prove that the algorithm is sub-factorial. Since most graphs don't seem to have sections, this means that existing algorithms based on partition refinement are already sub-factorial. It may be possible that the notion of sections can be extended to give a faster isomorphism algorithm.

This article appeared in *Computational and Constructive Design Theory*, edited by W.D. Wallis, Kluwer Academic Publishers, 1996.

2 PARTITION REFINEMENT

Let G be a graph. We assume that G is undirected, with no loops or multiple edges, but the algorithm can be fairly easily extended to apply to directed graphs and/or non-simple graphs and/or designs. The vertex and edge sets of G are $V(G)$ and $E(G)$. Each edge is an unordered pair of vertices. If $u, v \in V(G)$, we write the pair $\{u, v\}$ as uv. We write $u \to v$ to indica te that u is adjacent to v. Since G is undirected we also have $v \to u$. This can also be expressed as $uv \in E(G)$.

The graph isomorphism program we are describing produces a *certificate* for G, written cert(G). Two graphs G and H are isomorphic if and only if they have equal certificates, cert(G) = cert(H) (see Read and Corneil [26]). One way of defining a certificate (the commonest way) is this. Let G have n vertices. $A(G)$, the adjacency matrix of G, is a symmetric matrix with a diagonal of zeroes. Changing the ordering of the rows and columns will change the matrix $A(G)$. The upper triangle contains $\binom{n}{2}$ bits which can be written as a single binary number, row after row, or column after column. Each ordering of $V(G)$ defines a bit string in this way. Fig. 1 shows a graph G and an adjacency matrix $A(G)$. The bit string defined by taking the upper triangle of $A(G)$ column by column is then 1110010011010011000111001001. It is more concise to interpret these bit strings as character strings, by grouping them 6 or more bits at a time.

These bit strings can be ordered lexicographically, and the *smallest* (or largest) can be taken as cert(G). We say that cert(G) corresponds to the *smallest adjacency matrix* for G. When defined in this way, cert(G) is obviously independent of the original ordering of the vertices. The disadvantage is that there are $n!$ different orderings of $V(G)$. There are several ways of reducing this to something more manageable. We describe one way. If G contains vertices of different degree, we can first sort the vertices by degree, and consider only those orderings of $V(G)$ which preserve the degree of the vertices. This idea can be extended as follows.

An *ordered partition* Π of $V(G)$ is a list of *cells*, $\Pi = (C_1, C_2, \ldots, C_p)$, where each cell C_i is a set containing one or more vertices, $\cup_i C_i = V(G)$, and $C_i \cap C_j = \emptyset$, for $i \neq j$. The degree sequence of G defines an ordered partition of G. A partition Π is said to be *equitable* or *stable* if whenever $u, v \in C_i$, u and v are both joined to the same number of vertices of C_j, for all $j = 1, 2, \ldots, p$. If Π is not stable, then we can *refine* it, as defined by

This article appeared in *Computational and Constructive Design Theory*, edited by W.D. Wallis, Kluwer Academic Publishers, 1996.

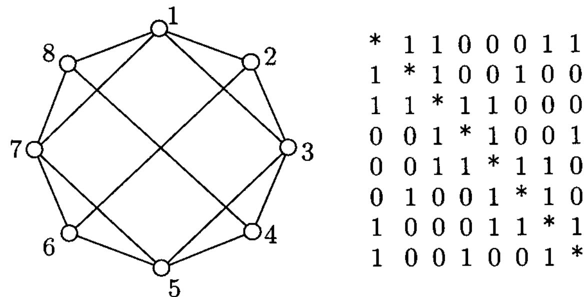

Figure 1 A graph and its adjacency matrix

the following algorithm. We assume that there is a global array *Degree*$[\cdot]$, such that *Degree*$[v]$ is a cell invariant for Π. Initially we can assume that

Degree$[v] = 0$, for all $v = 1, 2, \ldots, n$.

```
Refine(Π: partition)
{Π is an ordered list of cells}
{each cell of Π is initially marked uncounted}
Begin
    C := first cell of Π
    repeat
        {first count the adjacencies to cell C}
        for each u ∈ C do
            for each v → u do
                Degree[v] := Degree[v] + 1
        mark C counted
        for each cell C′ do begin
            sort the vertices of C′ by Degree[·]
            if Degree[·] is not constant on C′ then
                split C′ into new cells of equal degree
                all new cells created are marked uncounted
        end
        C := an uncounted cell of Π
    until Π is stable
End  {Refine}
```

This article appeared in *Computational and Constructive Design Theory*, edited by W.D. Wallis, Kluwer Academic Publishers, 1996.

For example, let us start with the *unit* partition Π_0 for the graph of Fig. 1. This is the unique partition with a single cell $C_0 = \{1,2,3,4,5,6,7,8\}$. The *Refine* algorithm first takes $C := C_0$ and counts the cell C. This has the effect of computing the degree of each vertex of G. The vertices of C are then sorted by degree. C contains vertices of degree 3 and 4. The next step sorts C and splits it into two cells, $C_1 = \{2,4,6,8\}$ of degree 3, and $C_2 = \{1,3,5,7\}$ of degree 4. So we now have $\Pi_1 = (C_1, C_2)$, where every $v \in C_1$ has *Degree*$[v] = 3$ and every $v \in C_2$ has *Degree*$[v] = 4$. In order to determine that Π_1 is in fact equitable, the repeat loop runs two more iterations. It first takes $C := C_1$ and counts this cell. Each $v \in C_1$ is adjacent to one vertex of C_1 and two vertices of C_2. Since *Degree*$[\cdot]$ is accumulative, the stored degrees of the vertices of C_1 will increase from 3 to 4, and those of C_2 will increase from 4 to 6. Since C_1 and C_2 still have constant degrees, there is no splitting of the cells at this point. C_1 is marked *counted*. The algorithm then takes $C := C_2$ and counts it. Each $v \in C_2$ is adjacent to two vertices of C_1 and two vertices of C_2. The stored degrees of the vertices of C_1 will increase from 4 to 6, and those of C_2 will increase from 6 to 8. Again there is no splitting of any of the cells. C_2 is now marked counted. Since every cell of Π_1 is counted, the algorithm deduces that the partition is stable, and the repeat loop terminates with an equitable partition. Thus an equitable partition is stable with respect to refinement.

The algorithm chooses C as the first uncounted cell of Π. It is obvious that with a specific choice like this, the equitable partition constructed will be independent of the original ordering of the vertices. In general, many ordered partitions correspond to the same unordered partition. If the cells C chosen to be counted are selected by a different rule, a different ordered partition will result. However it will always correspond to the same unordered partition. See Mathon [21] for a proof. A sketch of the proof follows. The set of all unordered partitions of V forms a partially ordered set. Π is smaller than Π' if it is finer than Π'. The set of partitions has a unique minimum element, the discrete partition, and a unique maximum element, the unit partition containing only one cell. Given any two partitions Π and Π', there is a unique partition $\Pi \wedge \Pi'$ which is the largest partition smaller than both Π and Π'. It is obtained by forming all pairwise intersections of the cells of Π and Π'. There is also a unique partition $\Pi \vee \Pi'$ which is the smallest partition larger than both Π and Π'. It is obtained by taking the unions of intersecting cells of Π and Π'. See Fig. 2. If we formed a bipartite graph of the cells of Π versus the cells of Π', and joined intersecting cells, then the cells of $\Pi \vee \Pi'$ would correspond to the connected components

This article appeared in *Computational and Constructive Design Theory*, edited by W.D. Wallis, Kluwer Academic Publishers, 1996.

of this graph. The cells of $\Pi \wedge \Pi'$ would correspond to the edges of this bipartite graph.

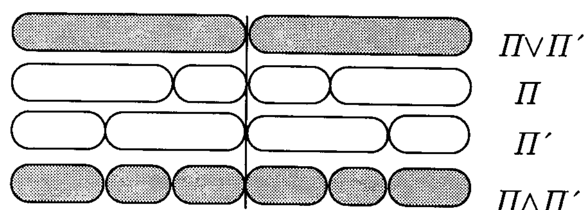

Figure 2 Partitions form an ordered set

Lemma 2.1 *Suppose that* Π *and* Π' *are equitable. Then so is* $\Pi \vee \Pi'$.

This can be proved by counting adjacencies using the diagram above. Note however, that $\Pi \wedge \Pi'$ is not always equitable. See McKay [25]. Now consider the refinement procedure in which a non-equitable partition Π is to be refined. A cell C of Π is selected and counted. This causes some of the cells of Π to split into two or more cells. Let Π_C be the equitable unordered partition obtained after refining. If the refinement were begun with another cell C' instead of C, an equitable unordered partition $\Pi_{C'}$ would be obtained. If $\Pi_C \neq \Pi_{C'}$, then $\Pi_C \vee \Pi_{C'}$ would be an equitable partition. Since $\Pi_C \leq \Pi$ and $\Pi_{C'} \leq \Pi$, it follows that $\Pi_C \vee \Pi_{C'} \leq \Pi$. But this is impossible. Thus for any unordered partition Π there is a unique largest equitable unordered partition Π_e that is smaller than Π. There are a number of equitable ordered partitions that correspond to Π_e. The refinement algorithm will construct one of them. It must choose the cell C to count by the same rule each time, in order to ensure that the ordering of Π_e that it constructs is unique.

The termination of the loop "repeat ... until Π is stable" requires that the program can recognize when a partition is stable. This will occur if every cell has been counted, and no further splitting of any cell has taken place. Whenever a new cell is created by splitting C', the new cells are marked uncounted. The program selects the cell C to count as an uncounted cell of Π. Consequently every cell will eventually be counted, and the loop will terminate only when no further splitting of cells has occurred. Therefore the repeat loop will terminate only when Π is equitable. So one way to recognize an equitable partition is to continue until every cell has been

This article appeared in *Computational and Constructive Design Theory*, edited by W.D. Wallis, Kluwer Academic Publishers, 1996.

counted. However one must keep in mind that eventually the isomorphism algorithm will construct a *discrete* partition of $V(G)$, that is, a partition whose every cell contains only one vertex. A discrete cell cannot split any further. Therefore *it is a good idea to remove discrete cells from the partition after they have been counted*, to avoid duplication of work. We will say more about discrete cells later. It is also possible that when a cell C' is split, that it splits into discrete cells. The entire partition Π can become discrete at such a step.

When this happens, there is no point in continuing to count the uncounted cells of Π, for a discrete partition must be equitable. So the criteria for detecting when Π is stable are:

1. if there is no uncounted cell C, then Π is stable;
2. if Π is discrete, then it is stable.

The first condition is very easy to detect. The second condition is not so easy. When the cell C' is being sorted according to *Degree*$[v]$, where $v \in C'$, the sorting algorithm can detect if all degrees of C' are distinct. If so, it can mark C' *discrete*, instead of actually splitting C' into new cells. A counter of the total number of vertices in discrete cells can be maintained. When it equals the number of vertices, we know that Π is discrete.

Data Structures

The *Refine* algorithm is tricky to program. The actual program will depend on the data structures chosen. In *Groups & Graphs*, I have used the following data structures. A graph is stored as an array of linked lists. For each vertex u, *Graph*$[u]$ denotes the list of vertices adjacent to u. This allows the loop "for all $v \to u$ do" to be programmed efficiently. An adjacency matrix is also stored, since it is needed for comparing different orderings of $V(G)$ when finding cert(G). The current ordering of the vertices is stored by an array $P[1..n]$. I have represented an ordered partition as a linked list of cells, where each cell is a record as follows.

```
CellPtr = ^Cell
Cell = record
        FirstPt, LastPt: Integer
        Counted, Discrete: Boolean
        NextCell: CellPtr
end
```

This article appeared in *Computational and Constructive Design Theory*, edited by W.D. Wallis, Kluwer Academic Publishers, 1996.

FirstPt and *LastPt* are pointers into the array $P[\cdot]$. The vertices in each cell are stored as contiguous entries in the array. *Counted* indicates whether the cell has been counted, and *Disrete* indicates whether a cell is composed of one or more discrete cells. *NextCell* is a pointer to the next cell in the partition.

This is a very convenient way to represent graphs and partitions, but it does have certain disadvantages. The main one is that a typical graph isomorphism calculation computes many thousands of partition refinements, and the constant creation and deletion of cells requires a great deal of memory management, which can be slow. Partition refinement is the main operation which takes place in graph isomorphism. Most of the time required is consumed by partition refinement. Therefore it must be made as fast as possible. McKay [22,24] has stored a graph as an adjacency matrix in such a way that each row is a packed bit-vector. This means that counting the cells in a partition can be accomplished using full-word boolean operations on the rows of the adjacency matrix. This counts all vertices adjacent to a vertex u in a constant number of steps (up to a maximum size). This gives a significant increase in speed. A difficulty which then arises is that it becomes more difficult to compare two orderings, since the adjacency matrix has been packed into bit vectors. On a parallel computer, partition refinement could take place much faster.

The algorithm *Refine*(Π) accepts an ordered partition Π and transforms it into an equitable ordered partition Π'. Viewed as unordered partitions, Π' is the largest equitable partition which is less than or equal to Π. The program assumes that initially *Degree*$[v]$ is a cell-invariant of Π (*Degree*$[v] = 0$ the first time *Refine* is called). When it terminates, Π has been transformed into an equitable partition Π', so that *Degree*$[v]$ will still be a cell-invariant of Π'. This feature is significant, because it means that the *program does not need to re-intiallize Degree*$[v]$ *to 0*. This saves an enormous amount of execution time.

Complexity

Suppose that a cell $C \in \Pi$ is to be counted, where $|C| = m$. If we assume that the graph is stored as adjacency lists, the number of steps required to count C is proportional to the sum of the degrees of the vertices of C. All cells in Π must then be sorted according to the array *Degree*$[\cdot]$. In theory a radix sort could be used to sort the cell in time $O(m)$, since the degrees are in the range $1..m$. However in practice this is not very useful, since it requires setting up m buckets, one for the vertices of each degree.

This article appeared in *Computational and Constructive Design Theory*, edited by W.D. Wallis, Kluwer Academic Publishers, 1996.

After the vertices have been placed in buckets, the buckets must be linked together, and this requires m steps, even if many of the buckets are empty. For example, m could be around 100, and there may be only 3 different degrees. Nevertheless, all 100 buckets would have to be created. Every vertex would have to be moved twice, once into a bucket, and once into its proper position. All 100 buckets would have to be tested to see if they are non-empty.

When a partition becomes equitable, all vertices in it have the same degree. It is extremely helpful to have a sort algorithm that only makes one pass through the vertices if they are already in order, leaving them unmoved. I have found that a simple insertion sort works very well for most graphs up to reasonable size. For larger graphs the quadratic complexity of the insertion sort starts to become noticeable. A very useful technique is to use a quicksort, with a cut-off M of around 16 to 32. When the number of vertices to be sorted is less than M, the quicksort procedure uses an insertion sort to order the vertices. When the number is M or more, a pivot is found and a recursive call is made. See Weiss [27] for an excellent implementation of quicksort.

Quicksort has an average complexity of $O(n \log n)$. Each time a cell is split the number of cells increases by at least one. If the number of vertices is n, the maximum number of splittings that can take place is n, giving a worst-case average complexity of $O(n^2 \log n)$ for the sorting stages of *Refine*. (The radix sort gives a theoretical worst-case complexity of $O(n^2)$, but it is not effective in practice.) If the graph has ε edges, the number of steps needed to count a cell C is at most $O(\varepsilon)$. If Π is already equitable, the total number of steps required to detect this is 2ε, by counting every cell. Since there are at most n splittings, the total number of steps used in counting cells is at most $O(n\varepsilon)$. Thus refinement is fairly efficient. As mentioned above, it is the single most time-consuming part of the algorithm. In order to make a polynomial graph isomorphism algorithm, a way must be found to reduce the number of discrete partitions to a polynomial number.

The Automorphism Group

The automorphism group of G consists of the set of all permutations of $V(G)$ that leave $E(G)$ invariant set-wise. It is denoted $\mathrm{Aut}(G)$. If $\pi, \tau \in \mathrm{Aut}(G)$, the image of vertex v under π is denoted v^π. Permutations are composed from left to right, so that $v^{\pi\tau}$ indicates the image of v under the product $\pi\tau$ (first π, then τ). A cell C of a partition Π is *fixed* by $\mathrm{Aut}(G)$ if $C^\pi = C$, for every $\pi \in \mathrm{Aut}(G)$, that is, π maps every element

This article appeared in *Computational and Constructive Design Theory*, edited by W.D. Wallis, Kluwer Academic Publishers, 1996.

of C to an element of C. If $\Pi = (C_1, C_2, \ldots, C_p)$, then Π^π denotes the ordered partition whose cells are $(C_1^\pi, C_2^\pi, \ldots, C_p^\pi)$. The connection between partition refinement and the automorphism group is the following.

Lemma 2.2 *Let Π_0 be a partition of $V(G)$ such that* $\mathrm{Aut}(G)$ *fixes every cell of Π_0. Let Π be obtained from Π_0 by refinement. Then* $\mathrm{Aut}(G)$ *fixes every cell of Π.*

Proof. The proof is by induction on the number of iterations of the repeat loop. Initially all automorphisms fix every cell of Π_0. On each iteration a cell C is counted. The vertices of each cell C' are then sorted according to the number of adjacencies they have to C, and C' may be split into two or more new cells. If $\pi \in \mathrm{Aut}(G)$ maps $u \in C$ to $v \in C$, then π also maps the vertices adjacent to u to the vertices adjacent to v. So π fixes every new cell split from C'. It follows that π fixes every cell of Π.

Since the unit partition is always fixed by $\mathrm{Aut}(G)$, for every graph G, we can always take Π_0 to be the unit partition initially, and refine it to an equitable partition Π. If H is any graph isomorphic to G and the same refinement procedure is applied to H, then any isomorphism from G to H must map Π, cell by cell, to the corresponding partition of H. The number of such mappings is no longer $n!$, but $\prod_{i=1}^{p} |C_i|!$, where $\Pi = (C_1, \ldots, C_p)$. A Π-*ordering* of G is any permutation of $V(G)$ that fixes every cell of Π setwise. The number of Π-orderings is $\prod_{i=1}^{p} |C_i|!$. We could then define $\mathrm{cert}(G)$ as the smallest adjacency matrix over the set of Π-orderings of $V(G)$. For example, if we compute *Refine*(Π_0) for the graph of Fig. 3, we find that the resulting equitable partition is discrete. This uniquely defines the certificate for G.

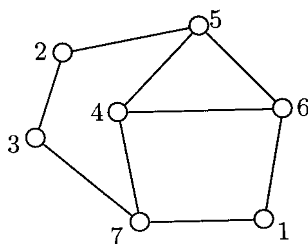

Figure 3 A graph with discrete partition

This article appeared in *Computational and Constructive Design Theory*, edited by W.D. Wallis, Kluwer Academic Publishers, 1996.

3 STABILISING VERTICES

If G is a regular graph, then the unit partition is already equitable, and refinement will produce no change in it. A number of recursive techniques can be used to reduce the number of orderings of $V(G)$ that must be considered in order to compute a certificate. The most common is vertex-stabilisation. Let Π be a non-discrete equitable partition of G. We can assume that Π has no discrete cells. Let Γ be a subgroup of $\mathrm{Aut}(G)$ such that Γ fixes Π. If C is any cell of Π, and $u \in C$ is any vertex of C, then in some of the Π-orderings of $V(G)$, u will be the first vertex of C. In order to focus on those orderings in which u is first, we split C into two cells, $\{u\}$ and $C - u$. Call the resulting partition Π'_u. Γ no longer fixes Π'_u, but the subgroup $\Gamma_u = \{\gamma \in \Gamma \mid u^\gamma = u\}$ that fixes u must also fix $C - u$ and all other cells of Π'_u.

3.1 Let Γ be a permutation group acting on a set V. The subgroup Γ_u consisting of those permutations that fix u is called a *stabiliser* subgroup.

We now refine Π'_u to an equitable partition Π_u. By Lemma 2.2, Γ_u fixes Π_u. This process of fixing a vertex u in a partition Π and refining Π'_u to an equitable partition Π_u is called *vertex stabilisation*. In most graphs it produces a considerable refinement of Π.

3.2 Given a vertex $u \in C \in \Pi$, Π_u always denotes the equitable partition obtained by splitting C and refining the resulting partition.

The number of Π_u-orderings of V will be much less than the number of Π-orderings. Since some $u \in C$ must be first, we do this for each $u \in C$, and then apply this idea recursively. This gives the following skeleton of a recursive procedure. The algorithm saves the ordering of V that gives the smallest adjacency matrix so far. This is called the best ordering.

```
Stabilise(Π: partition) {first version}
{refine Π to an equitable partition, with no discrete cells. If it is
  not discrete, fix a point from the first cell in all possible ways.}
Begin
    Refine(Π)
    if Π is discrete then begin
```

This article appeared in *Computational and Constructive Design Theory*, edited by W.D. Wallis, Kluwer Academic Publishers, 1996.


```
            Π defines an ordering of the vertices
            compare it with the best ordering found so far,
              replacing the best if necessary
            return
        end
        {otherwise Π is not discrete}
        C := first cell of Π
        for each u ∈ C do begin
            make a copy Π_u of Π in which C is split into
                {u} and C − {u}
            Stabilise(Π_u)
            Dispose(Π_u)
            {since some vertex of C must be first, and we have tested
                all vertices of C, at this point we have cert(G)}
        end
    End {Stabilise}
```

The calling program begins by setting up a unit partition of V and calling *Stabilise*(Π). The recursion defines a search tree. A *leaf* in a search tree is a node with no descendants. Each leaf of the search tree corresponds to a discrete partition of V. Each discrete partition defines an ordering of V. The certificate of G is now *defined* as the smallest adjacency matrix with respect to this restricted set of orderings. Any ordering that gives the smallest adjacency matrix is called a *canonical* ordering. The program maintains a global array $B[1..n]$ of the best ordering of V found so far. This is the ordering that gives the smallest adjacency matrix. When the program terminates B will be a canonical ordering. It also maintains a global array $V[1..n]$ of the current ordering of the vertices, as constructed by partition refinement. At this point we need to say a word about the discrete cells produced by refinement. When *Refine*(Π) is executed, one or more discrete cells may be produced. Once a discrete cell has been counted, we remove it from Π, since a discrete cell cannot split any further. We place these deleted vertices on a third global array $F[1..n]$ of *fixed* points, *in the order in which they are encountered.* It may be that 5 or 6 points become fixed during a refinement. The number of fixed points currently on F is stored in a variable *NFixed*. We use the array F to compare with B. We compare the entries in the upper triangle of $A(G)$ as shown below. Comparing two orderings is quite time consuming, since it requires a lot of access to the 2-dimensional array A. The function compares F and B between subscripts

This article appeared in *Computational and Constructive Design Theory*, edited by W.D. Wallis, Kluwer Academic Publishers, 1996.

i and j, which will be $i = 1$ and $j = n$ when orderings arising from discrete partitions are compared.

```
Function Compare(i, j: integer): integer
{compare the orderings of A defined by arrays F and B
  between entries i and j. Returns -1, 0, or 1 according
  as F is worse than, equivalent to, or better than B}
begin
    if i = 1 then i := 2   {nothing to compare in column 1}
    for c := i to j do   {column c}
        for r := 1 to c - 1 do begin   {row r}
            if A[F[r], F[c]] < A[B[r], B[c]] then return(1)
            if A[F[r], F[c]] > A[B[r], B[c]] then return(-1)
        end
    return(0)
end   {Compare}
```

We now define cert(G) as the smallest adjacency matrix with respect to the orderings produced by *Stabilise*(Π). For most graphs there will be much less than $n!$ orderings. In fact, in a probabilistic sense, for "almost all" graphs, the partition obtained by refining the unit partition will be discrete [2]. However, most interesting graphs require more work.

Let us consider the complete graph K_n. All the partitions obtained by fixing a vertex are equitable. Refinement produces no improvement. The depth of the recursion will be n, corresponding to 0, 1, 2, ..., $n-1$ vertices being fixed. Clearly the search tree contains $n!$ leaf nodes; but they obviously all give the same certificate, so it is not really necessary to search them all. (Actually K_n can easily be detected since it has $\binom{n}{2}$ edges.)

3.3 If π is a permutation of V, A^π denotes the adjacency matrix obtained from A by permuting the rows and columns by π. Two orderings π_1 and π_2 of V are called *equivalent* if $A^{\pi_1} = A^{\pi_2}$. Notice that in this case, $A^{\pi_1\pi_2^{-1}} = A$, so that $\pi_1\pi_2^{-1} \in \mathrm{Aut}(G)$.

Lemma 3.1 *The number of inequivalent orderings of V is $n!/|\mathrm{Aut}(G)|$.*

This article appeared in *Computational and Constructive Design Theory*, edited by W.D. Wallis, Kluwer Academic Publishers, 1996.

Proof. Two orderings π_1 and π_2 are equivalent if and only if $\pi_1\pi_2^{-1} \in \mathrm{Aut}(G)$. So each coset of $\mathrm{Aut}(G)$ in the symmetric group $\mathrm{Sym}(V)$ gives a set of equivalent orderings.

In particular, K_n has only one (inequivalent) ordering of the vertices. If G has no automorphisms, then there are $n!$ inequivalent orderings of V. Any two leaf nodes of the search tree are inequivalent. Notice that this does not mean that the search tree has $n!$ inequivalent leaf nodes. I don't think anyone has been able to accurately estimate the number of leaf nodes of the search tree. Corneil and Goldberg [9] have shown that the number is sub-factorial, but this is still a weak bound. A proof that there is a polynomial number would give a polynomial algorithm for graph isomorphism. However the behaviour of all current graph isomorphism algorithms on certain difficult graphs suggests that they are not polynomial in general. The difficult graphs are invariably those that have very few automorphisms, but a high degree of "regularity", by which I mean that they have a large number of equitable partitions – the cell-adjacencies are very regular in an equitable partition. In other words, the regularity makes a graph look as though it were symmetric, when fact it may not be. The search tree will then contain a very large number of inequivalent leaf nodes.

Permutation Groups

We will need a number of basic properties of permutation groups and stabilisers. They are listed below. They are easy to verify from first principles. Here Γ is a permutation group acting on V, π is any element of Γ, u is any element of V and $v = u^\pi$. The order of a group Γ is denoted $|\Gamma|$. See the books by Biggs and White [5] or Wielandt [28] for more information on permutation groups.

3.4 The coset $\Gamma_u\pi$ consists of all elements of Γ that map u to $v = u^\pi$.

3.5 The *orbit* of u is the set of all points that Γ maps u to, that is, $\mathrm{Orb}(u) = u^\Gamma = \{u^\gamma \mid \gamma \in \Gamma\}$. It follows from 3.4 that $|u^\Gamma|.|\Gamma_u| = |\Gamma|$.

3.6 The *conjugate* of a permutation $\gamma \in \Gamma$ by π is $\pi^{-1}\gamma\pi$, which is also denoted by γ^π. If γ maps u to w, then γ^π maps u^π to w^π.

3.7 The conjugate of the subgroup Γ_u is $\pi^{-1}\Gamma_u\pi = \Gamma_u^\pi = \Gamma_v$, that is, the conjugate of the stabiliser of u by π is the stabiliser of $v = u^\pi$.

This article appeared in *Computational and Constructive Design Theory*, edited by W.D. Wallis, Kluwer Academic Publishers, 1996.

3.8 If Γ_u fixes a partition Π, then Γ_u^π fixes Π^π.

Lemma 3.2 *Suppose that $\Gamma \leq \mathrm{Aut}(G)$ fixes Π, and let Π_u be obtained by partition refinement. If π is any element of Γ, let $v = u^\pi$. Then $\Pi_v = \Pi_u^\pi$.*

Proof. Let $\Pi = (C_1, C_2, \ldots, C_p)$. Then $\Pi^\pi = (C_1^\pi, C_2^\pi, \ldots, C_p^\pi)$. If $u \in C_i$, then $u^\pi \in C_i^\pi = C_i$. When Π_u is constructed by refinement, a number of adjacencies in the graph G are counted. Since π is an automorphism of G, it follows that if a vertex w is adjacent to k vertices of a cell C_j of Π, that w^π will also be adjacent to k vertices of $C_j^\pi = C_j$ in Π^π, and so forth. Thus at every step of the refinement process, the cells of Π_u can be mapped by π to those of Π_v. When the process terminates, $\Pi_u^\pi = \Pi_v$. Notice that Γ_u fixes Π_u and Γ_u^π fixes Π_u^π.

In general let $\Pi_0, \Pi_1, \ldots, \Pi_k$ be the sequence of partitions occurring in a path to a leaf node in the search tree. Π_0 is the initial equitable partition obtained by refining the unit partition. A vertex u_0 in Π_0 is fixed, and after refinement, Π_1 is obtained. Then u_1 in Π_1 is fixed, and so on, until the discrete partition Π_k is reached. We assume throughout that discrete cells are deleted as they are encountered, so that $\Pi_0, \Pi_1, \ldots, \Pi_{k-1}$ do not have any discrete cells. Only Π_k has discrete cells. Vertices $u_0, u_1, \ldots, u_{k-1}$ have all been fixed in succession. As outlined above, the program *Stabilise*(Π) always selects u in the *first* cell C of Π. The reason for this is that the first cell is very easy to find. The program does not have to do any work to select it. Other strategies are possible. We could select u in the smallest cell, or largest cell, or some other specific choice. This would perhaps make a smaller or larger search tree, but requires more work to select u. So let C_i denote the first cell of Π_i. Then $u_i \in C_i$, for $i = 0, 1, \ldots, k-1$. Let $\pi \in \mathrm{Aut}(G)$, and let $v_i = u_i^\pi$, for $i = 0, \ldots, k-1$. Then $\Pi_0^\pi = \Pi_0$. Fixing v_0 instead of u_0 gives the partition Π_1^π, by Lemma 3.10. Since $u_1 \in C_1$, we know that $v_1 \in C_1^\pi$, the first cell of Π_1^π. Fixing v_1 in Π_1^π gives the partition Π_2^π, and so on. Thus, fixing $u_0^\pi, u_1^\pi, \ldots, u_{k-1}^\pi$ gives a sequence $\Pi_0^\pi, \Pi_1^\pi, \ldots, \Pi_k^\pi$ of partitions, such that the discrete partition Π_k^π is equivalent to Π_k.

A sequence $u_0, u_1, \ldots, u_{k-1}$ of points fixed in order to produce a discrete partition Π_k is termed a *basis* for the ordering of V defined by Π_k. The search tree constructed by *Stabilise* can be defined as the set of all sequences $(u_0, u_1, \ldots, u_j)$ of points fixed in succession by the algorithm, with adjacencies determined by *descent*: the sequence $(u_0, \ldots, u_j, u_{j+1})$ decends

This article appeared in *Computational and Constructive Design Theory*, edited by W.D. Wallis, Kluwer Academic Publishers, 1996.

from $(u_0, \ldots, u_j)$. Each descent corresponds to the vertex fixed, in this case, u_{j+1}. The tree is rooted at the empty sequence (), corresponding to no points fixed. Each sequence $(u_0, \ldots, u_j)$ corresponds to the equitable partition Π_{j+1} constructed by fixing these points in succession. The root node corresponds to Π_0. Each leaf node corresponds to a discrete partition. The path from the root to a leaf node defines the basis for that ordering of V. Fig. 4 shows a graph whose search tree is illustrated in Fig. 7. Each node of the search tree contains the corresponding partition. In this example, the discrete cells of each partition have not been deleted. Each descent is labelled by the vertex fixed at that point in the algorithm.

The remarks of the preceding paragraph show that if $\pi \in \text{Aut}(G)$, then π induces an automorphism of the search tree. We know that $\text{Aut}(G)$ fixes Π_0. The algorithm selects u_0 in Π_0 and fixes it to get Π_1. The branch of the search tree descending from Π_0 on the edge labelled u_0 will be searched by *Stabilise*(Π_1). If u_0^π in Π_0 is fixed instead, to get Π_1^π, then the branch descending on the edge labelled u_0^π will be isomorphic to the branch just searched, by Lemma 3.10. So having fixed u_0, we need fix no other points u_0^π for any $\pi \in \text{Aut}(G)$. Write $\Gamma_0 = \text{Aut}(G)$. Let Γ_1 denote the stabiliser of u_0 in Γ_0. Then Γ_1 fixes Π_1. In general, let Γ_i denote the subgroup of Γ_{i-1} obtained by stabilising u_{i-1}. Then Γ_i fixes Π_i. The algorithm selects some $u_i \in C_i \in \Pi_i$ and fixes it. Let $\pi \in \Gamma_i$. The branch of the search tree descending from Π_i on the edge labelled u_i will be isomorphic to the branch descending on the edge labelled u_i^π. So having chosen $u_i \in C_i$, we need choose no other points u_i^π for any $\pi \in \Gamma_i$. That is, we need to fix *exactly one point from each orbit* of Γ_i on C_i in order to find the unique minimum adjacency matrix of G.

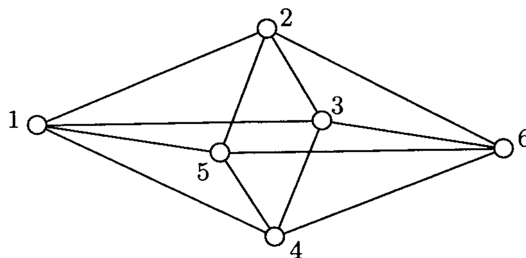

Figure 4 The graph of the octahedron

Initially the algorithm does not know $\text{Aut}(G)$ or any of the stabiliser subgroups Γ_i. However as the program executes it will occasionally find a leaf node whose ordering of V is equivalent to the best ordering. By Lemma

This article appeared in *Computational and Constructive Design Theory*, edited by W.D. Wallis, Kluwer Academic Publishers, 1996.

3.4 this means that an automorphism of G has been discovered. The program maintains a data structure representing $\Gamma_0, \Gamma_1, \Gamma_2, \ldots$. Each time an automorphism π is discovered, the data structure is updated. This enables the program to avoid searching most equivalent branches of the tree. In the next section we describe the data structures used to represent the stabiliser subgroups.

4 THE SCHREIER-SIMS ALGORITHM

Given a set of permutations of a set V, we can make a *permutation diagram* for them. It is a directed graph in which the vertices are the elements of V. The edges are labelled by the permutations. If π is a permutation that maps u to $v = u^\pi$, then there is an edge from u to v labelled π. Fig. 5 shows a permutation diagram for $\pi = (1, 6, 5, 2, 4, 3)$ and

$\tau = (1, 4, 3)(2, 5)(6)$. These diagrams are the basis of many algorithms for groups.

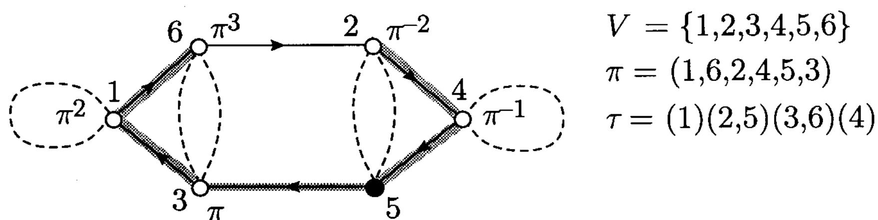

Figure 5 A permutation diagram

Let Γ denote the group generated by π and τ. Suppose that we are given π and τ and want to find Γ and a sequence of stabiliser subgroups. Γ consists of all products that can be formed from π, τ and their inverses. It is very easy to construct the permutation diagram for π and τ. Notice that each connected component of the permutation diagram represents an orbit of Γ. In the example above there is only one orbit. Suppose that we start at vertex 1 in the graph of Fig. 5 and follow the edges labelled $\pi\pi\tau\pi\tau\pi^{-1}$. We come to vertices 1,6,2,5,3,6,1, in that order. We conclude that the product $\pi\pi\tau\pi\tau\pi^{-1}$ maps 1 to 1, that is, it is an element of the stabiliser of 1 in Γ. Thus we have the following important observation.

This article appeared in *Computational and Constructive Design Theory*, edited by W.D. Wallis, Kluwer Academic Publishers, 1996.

4.1 Every walk in the permutation diagram corresponds to a product of the generators and/or their inverses. Given any starting point $u \in V$, every product of the generators and/or their inverses corresponds to a walk in the permutation diagram starting at u. If the walk ends at point $v \in V$, then u is mapped to v by this product.

Thus the stabiliser Γ_u of a point u corresponds to all walks that start and end at u. The walks that start at u and end at v correspond to all elements of Γ mapping u to v, that is, a right coset of Γ_u (see 3.5). Let us choose a spanning tree T of the graph, and pick any point u as the *root node* of T. Fig. 5 shows a spanning tree T in grey rooted at point 5. Now T contains a unique path from u to every point in V. These paths define a representative element of Γ mapping u to v, for every v. This gives a second important observation.

4.2 Every spanning tree T of a connected component X of the permutation diagram defines a decomposition of Γ into right cosets of Γ_u, where u is any point in X.

The spanning tree in Fig. 5 is rooted at point 5. The cosets of Γ_5, the stabiliser of point 5, are then $\Gamma_5\pi$, $\Gamma_5\pi^2$, $\Gamma_5\pi^3$, $\Gamma_5\pi^{-1}$, and $\Gamma_5\pi^{-2}$, as determined by T. If we could construct generators for Γ_u, then together with the coset representatives, this would completely determine Γ.

Let X denote a connected component of the permutation diagram with spanning tree T rooted at u. For every vertex v in X let T_v denote the word in the generators corresponding to the path in T from u to v. If vw is any edge not in T, then $T+vw$ contains a unique cycle, the *fundamental cycle* C_{vw} of vw with respect to T. Suppose that vw corresponds to a generator π. Then $T_v\pi T_w^{-1}$ is a walk in X that starts at u, travels to v, follows the edge vw to w, then travels along T_w^{-1} back to u. That is, it is a walk that starts at u, travels along a path in T to the fundamental cycle of vw, follows the cycle around, and then returns to u. See Fig. 6. In Fig. 5, for example, if the edge $(6,3)$ corresponding to the generator τ is used, the fundamental cycle is $(3,1,6)$ and the walk $T_6\tau T_3^{-1}$ gives the sequence $(5,3,1,6,3,5)$ of vertices, which corresponds to the product $\pi\pi\pi\tau\pi^{-1}$ of the generators. Any edge vw not in T is called a *chord* of T.

It is a theorem of graph theory that every closed walk in a graph can be decomposed into fundamental cycles. See Bollobas [6] for a proof. We

This article appeared in *Computational and Constructive Design Theory*, edited by W.D. Wallis, Kluwer Academic Publishers, 1996.

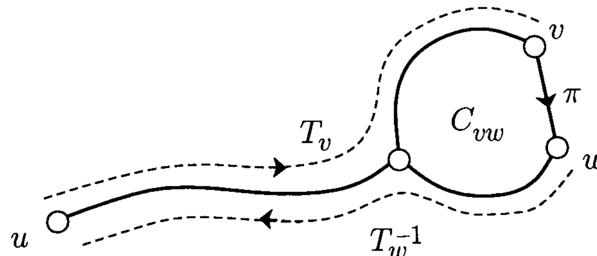

Figure 6 A fundamental cycle C_{vw}

sketch an outline of the proof here. Every cycle C in X must contain at least one edge not in T. Suppose that it contains edges $v_1w_1, v_2w_2, \ldots, v_kw_k$ not in T. If $k = 1$ then C is the fundamental cycle $C_{v_1w_1}$. $T_{v_1}v_1w_1T_{w_1}^{-1}$ is a walk that starts at u, travels to C, travels around C and then returns to u. If $k > 1$ we prove by induction that $T_{v_1}v_1w_1T_{w_1}^{-1} \ldots T_{v_k}v_kw_kT_{w_k}^{-1}$ is a closed walk that starts at u, travels to C, travels around C and returns to u. To complete the proof it is only necessary to show that every closed walk can be decomposed into cycles.

Since a walk that starts and ends at u is an element of the stabiliser Γ_u, we have another important observation. This result is of fundamental importance to many algorithms for groups.

4.3 Let T be a spanning tree of the permutation diagram, with root node u. Then Γ_u is generated by the products of the generators given by the walks $T_vvwT_w^{-1}$, where vw is any chord of T.

A consequence of 4.3 is that we now have a method of constructing a representation of any permutation group Γ for which generators are given. It is called the Schreier-Sims algorithm [7,8].

We store a permutation of V as an array of integers. The number of points in V will only be known when the program executes. Therefore we store pointers to permutations and allocate the arrays dynamically with the appropriate length. When storing a group, we need an array of coset representatives, that is, an array of permutations. Since we don't know the length of the array at compile-time, we store a pointer to an array of pointers. The generators of a group are stored as a linked list of permutations.

This article appeared in *Computational and Constructive Design Theory*, edited by W.D. Wallis, Kluwer Academic Publishers, 1996.


```
PermPtr = ^Perm
Perm = array[1..n] of integer
CosetPtr = ^CosetReps
CosetReps = array[1..n] of PermPtr
GenPtr = ^Gen
Gen = record
        GenPerm: PermPtr
        NextGen: GenPtr
    end
```

The data structure used to represent a permutation group Γ contains a linked list of generators, an orbit Orb(u), for some point $u \in V$, and a representative for each coset of Γ_u. It is also convenient to store the inverses of some of the coset representatives. The orbit is stored on an array, which is allocated dynamically as a PermPtr. The number of points in the orbit is stored as *NPts*. The data structure also contains the stabiliser Γ_u. Since Γ_u is also a group, we have a recursive data structure.

```
GroupPtr = ^Group
Group = record
        Generators: GenPtr  { linked list of generators }
        u: integer
        Orbit: PermPtr  { the orbit of u }
        NPts: integer  { number of points in Orb(u) }
        Cosets: CosetPtr  { coset reps of Γ_u }
        Inverses: CosetPtr
        Γ_u: GroupPtr  { stabiliser of u }
    end
```

Orbit^$[k]$ is the kth point in the orbit of u, for $k = 1, 2, \ldots,$ *NPts*. The first point in the orbit is *Orbit*^$[1] = u$. The coset representative for $v =$*Orbit*^$[k]$ is *Cosets*^$[v]$. If v is a point not in the orbit of u, then *Cosets*^$[v]$ = *nil*. Given this representation of a group Γ acting on $V = \{1, 2, \ldots, n\}$, and any permutation γ on V we can easily determine whether $\gamma \in \Gamma$, as follows.

```
GroupElt(γ: PermPtr; Γ: GroupPtr): Boolean
{returns true if γ ∈ Γ}
var π: PermPtr
Begin
```

This article appeared in *Computational and Constructive Design Theory*, edited by W.D. Wallis, Kluwer Academic Publishers, 1996.


```
    for k := 1 to n do if γ^[k] ≠ k then goto 1
    return(true)   {γ ∈ Γ since γ is the identity}
1: with Γ^ do
    begin
        v := γ^[u]
        if Cosets^[v] =nil then return(false)      {v ∉ Orb(u)}
        π := MultiplyPerm(γ, Inverses^[v])
                                  {π maps u to u}
        return(GroupElt(Γu, π))
    end
End   {GroupElt}
```

The program first checks whether γ is the identity. We assume that the number of points, n, is a global variable. If γ is not the identity, $v = u^\gamma$ is found, and the program checks whether $v \in$Orb(u). If not, then $\gamma \notin \Gamma$. Otherwise let γ_v denote the coset representative stored for the point v ($\gamma_v =$ Cosets^$[v]$). The product $\pi = \gamma\gamma_v^{-1}$ is computed. Clearly π fixes u, so that $\gamma \in \Gamma$ if and only if $\pi \in \Gamma_u$. Therefore the program makes a recursive call. A function *MultiplyPerm* is needed in order to compute the product $\gamma\gamma_v^{-1}$. This assumes that the inverses of all coset representatives have been stored. If this is not the case, an additional statement is needed:

```
if Inverses^[v] = nil then Inverses^[v] := InversePerm(Cosets^[v])
```

where *InversePerm* is the following function.

```
InversePerm(γ: PermPtr): PermPtr
var τ: PermPtr
begin
    τ := NewPerm(n)   {allocate a new perm on n points}
    for k := 1 to n do τ^[γ^[k]] := k
    return(τ)
end   {InversePerm}
```

We are now in a position to give the code for the Schreier-Sims algorithm for constructing a representation of a permutation group Γ from its generators. The algorithm is based on a breadth first search to construct the permutation diagram for the generators of Γ. It builds a breadth-first spanning tree T which is used to define a set of coset representatives for the stabiliser

This article appeared in *Computational and Constructive Design Theory*, edited by W.D. Wallis, Kluwer Academic Publishers, 1996.

subgroup Γ_u. The chords of the spanning tree are used to construct generators for Γ_u. The main procedure *AddGen* takes an existing group Γ which is already stored. The first time it is called, Γ will have been intiallized to the identity group on n points. This is a *Group* record whose arrays have been allocated to have length n, but with no generators, and no point u selected. In an identity group the arrays *Cosets* and *Inverses* are allocated and every entry is intiallized to *nil*, and the stabiliser subgroup Γ_u is set to *nil*. The program takes a permutation γ which is a generator of Γ, but not yet represented in the data structure. Thus, $\gamma \notin \Gamma$ when the procedure is called. It updates the data structure so that upon completion γ is now recognized as a generator of Γ.

```
AddGen(γ: PermPtr; Γ: GroupPtr)
{add γ to the data structure representing Γ}
var π: PermPtr
Begin with Γ^ do begin
    if Γu = nil then begin
        {initialize Γu to the identity group on n points}
        Γu := IdentityGroup(n)
        {find a point u moved by γ}
        u := 1
        while γ^[u] = u do u := u + 1
    end
    add γ to the linked list Generators
    M := NPts   {current number of points in Orb(u)}
    k := 1
    while k ≤ M do begin
        v := Orbit^[k]   {k^th point in the orbit}
        w := γ^[v]
        if Cosets^[w] = nil then begin   {w ∉ Orb(u)}
            NPts := NPts + 1
            Orbit^[NPts] := w   {add w to Orb(u)}
            Cosets^[w] := MultiplyPerm(Cosets^[v], γ)
        end
        else begin   {w ∈ Orb(u)}
            if Inverses^[w] = nil then
                Inverses^[w] := InversePerm(Cosets^[w])
            π := GenMultiply(Cosets^[v], γ, Inverses^[w])
```

This article appeared in *Computational and Constructive Design Theory*, edited by W.D. Wallis, Kluwer Academic Publishers, 1996.


```
                if not GroupElt(π, Γ_u) then AddGen(π, Γ_u)
                else Dispose(π)   {π is not needed}
            end
            k := k + 1
        end
        {if γ has extended Orb(u), apply all generators to all new points}
        while k ≤ NPts do begin
            v := Orbit^[k]   {k^th point in the orbit}
            for each generator τ ∈ Generators do begin
                w := τ^[v]
                if Cosets^[w] = nil then begin   {w ∉ Orb(u)}
                    NPts := NPts + 1
                    Orbit^[NPts] := w   {add w to Orb(u)}
                    Cosets^[w] := MultiplyPerm(Cosets^[v], τ)
                end
                else begin   {w ∈ Orb(u)}
                    if Inverses^[w] = nil then
                        Inverses^[w] := InversePerm(Cosets^[w])
                    π := GenMultiply(Cosets^[v], τ, Inverses^[w])
                    if not GroupElt(π, Γ_u) then AddGen(π, Γ_u)
                    else Dispose(π)   {π is not needed}
                end
                k := k + 1
            end
        end
    end end   {AddGen}
```

The procedure first computes $w = v^\gamma$ for all $v \in \text{Orb}(u)$. Let $\gamma_v =$ Cosets^[v] be the coset representative for v. If $w \notin \text{Orb}(u)$, then w is added to the orbit. $\gamma_v\gamma$ maps u to w, so it becomes γ_w, the coset representative for w. It is computed by *MultiplyPerm*. If $w \in \text{Orb}(u)$, then $\pi = \gamma_v\gamma\gamma_w^{-1}$ maps u to u. Therefore it is a generator for the stabiliser Γ_u. *GenMultiply* is a procedure that performs this multiplication. This is more efficient than calling *MultiplyPerm* twice.

```
GenMultiply(α, β, γ: PermPtr): PermPtr
{compute αβγ}
var τ: PermPtr
```

This article appeared in *Computational and Constructive Design Theory*, edited by W.D. Wallis, Kluwer Academic Publishers, 1996.


```
begin
    τ := NewPerm(n)   {allocate a new perm on n points}
    for k := 1 to n do τ^[k] := γ^[β^[α^[k]]]
    return(τ)
end  {GenMultiply}
```

The program calls *GroupElt* to check if π is known to be in Γ_u. If so it is discarded. The storage used by it is reclaimed by *Dispose*, an operating system call. If π is not currently known to be in Γ_u then *AddGen* is called recursively. When it returns, the data structure representing Γ_u will have been modified to account for π. Once γ has been applied to all $v \in \mathrm{Orb}(u)$, the program applies all generators of Γ to all new points in the orbit. This builds the permutation diagram for the orbit containing u. When the program terminates, a coset representative has been stored for each $v \in \mathrm{Orb}(u)$, and the stabiliser Γ_u is up to date. Therefore the data structure for Γ is also up to date.

The sequence of points fixed in order to construct the tower of stabilisers is called the *basis* of the group. See Butler and Cannon [7] and Butler and Lam [8] for further information. The program *AddGen* constructs the permutation diagram using a breadth-first search. A number of variations on this algorithm are possible. See Kirk [17], Butler and Cannon [7] and Butler and Lam [8]. If there are k generators and n points in the orbit of u, the number of coset representatives stored will be n. The number of steps required to multiply two permutations of degree n is n. Therefore the number of steps required to build the diagram is proportional to kn^2, for one level in the recursion. The depth of the recursion will depend on the group Γ. The symmetric group requires depth n. The alternating group requires depth $n-1$. All other groups require much less, since any group which is 6-transitive or more must be either an alternating or symmetric group. The number of generators can always be taken to be at most $\log_2 |\Gamma|$, since each new generator must expand the tower by at least doubling the size of at least one group. Since $|\Gamma| \leq n!$, we can use Stirling's formula $n! \approx n^n e^{-n}\sqrt{2\pi n}$ to get a polynomial bound on the complexity of *AddGen*. However the only groups that reach the bound are the alternating and symmetric groups. These can often be detected and handled as special cases to give a much better complexity. A number of variants are also possible for the *AddGen* procedure such as randomized methods for building the group, or storing the coset representatives as words in the generators, rather than permutations.

This article appeared in *Computational and Constructive Design Theory*, edited by W.D. Wallis, Kluwer Academic Publishers, 1996.

5 GRAPH ISOMORPHISM

The procedure *AddGen* is called each time an automorphism is discovered in the search tree created by *Stabilise*. Notice that the automorphism group Γ is stored as a set of coset representatives plus a stabiliser subgroup. Together these create a tower of stabilisers. Each time an automorphism is discovered the effect is to expand the entire tower. Therefore the orbits will be updated for each group in the tower.

In section 3 we had a sequence of partitions $\Pi_0, \Pi_1, \ldots, \Pi_k$ and a sequence of vertices $u_0, u_1, \ldots, u_{k-1}$ such that u_i in Π_i was stabilised in order to obtain Π_{i+1}. A sequence of groups $\Gamma_0, \Gamma_1, \ldots, \Gamma_k$ is associated with the partitions. $\Gamma_0 = \text{Aut}(G)$. Γ_{i+1} is the stabiliser of u_i in Γ_i, where $i = 0, 1, \ldots, k-1$. Γ_i fixes Π_i. Γ_k is the identity group. So the basis $u_0, u_1, \ldots, u_{k-1}$ of the discrete partition Π_k is also a basis for the group Γ_0. Associated with each node of the search tree is a partition Π_i and a group Γ_i. Associated with each descent from Π_i to Π_{i+1} is the vertex u_i. u_i is contained in the first cell C_i of Π_i. In order to select a vertex from each orbit of Γ_i acting on C_i we must store the orbits of Γ_i on C_i. They are stored in an array referenced by the pointer *CellOrbits*. We collect together in one data structure the various objects associated with a partition.

```
SearchNodePtr = ^SearchNode
SearchNode = record
    Partition: CellPtr  {a partition Π_i}
    FixedPt: Integer  {the point u_i fixed to get Π_{i+1}}
    ItsGroup: GroupPtr  {the group Γ_i}
    CellOrbits: PermPtr  {the orbits of Γ_i on C_i}
    Depth: Integer  {current depth of the search tree}
    NFixed: Integer  {the number of fixed points on the array F}
    OnBestPath: Boolean  {whether the node is on the best path}
    NextSearchNode: SearchNodePtr  {the next search node, descending}
  end
```

The Cell Orbits.

The orbits of Γ_i on C_i, the first cell of Π_i, are stored using the *union-find* data structure (see Aho, Hopcroft and Ullmann [1], Weiss [27]). *CellOrbits* points to an integer array of length $n = |V|$. Each orbit has a representative vertex. Two vertices are in the same orbit if and only if they have the

This article appeared in *Computational and Constructive Design Theory*, edited by W.D. Wallis, Kluwer Academic Publishers, 1996.

same representative. The value of *CellOrbits*^$[v]$ is a pointer toward the representative, which in turn is marked by a negative value. We initiallize *CellOrbits*^$[v] = -1$, for all $v \in C_i$. The entries for $v \notin C_i$ are not used. The value of -1 indicates that each v is an orbit representative, and thus forms an orbit by itself. This is the situation initially when Γ_i is the identity group. When a generator γ of Γ_i is found, there will be a call to *AddGen*(γ, Γ_i). *AddGen* must update the known orbits of Γ_i on C_i. Thus it will contain a call to the following procedure, which can be placed immediately after the statement adding γ to the list of generators.

```
UpdateOrbits(γ: PermPtr, C: CellPtr)
{update the orbits of the stabiliser on cell C}
Begin with C^ do begin
    for k := FirstPt to LastPt do begin
        u := V[k]   {k^th vertex of the graph}
        v := γ^[u]
        uRep := OrbitRep(u)   {orbit representative for u}
        vRep := OrbitRep(v)   {orbit representative for v}
        if uRep ≠ vRep then Merge(uRep, vRep)
    end
end End   {UpdateOrbits}
```

In order to have access to the partition we must change the calling parameters of *AddGen* to include the current searchnode, which in turn contains the group.

```
AddGen(γ: PermPtr; S: SearchNodePtr)
```

The procedure *UpdateOrbits* computes $v = u^\gamma$ for all $u \in C$ and merges the orbits of u and v if they are different. Orbits are merged by reassigning the *CellOrbits* pointer for one of u and v. For efficiency, we merge the smaller orbit onto the larger (see [1]). The size of an orbit is given by the negative value stored in *CellOrbits* for the orbit representatives.

```
Merge(uRep, vRep: Integer)
{merge the orbits of uRep and vRep, which are orbit representatives}
Begin
    uSize := −CellOrbits^[uRep]
    vSize := −CellOrbits^[vRep]
    if uSize < vSize then begin
```

This article appeared in *Computational and Constructive Design Theory*, edited by W.D. Wallis, Kluwer Academic Publishers, 1996.


```
            CellOrbits^[uRep] := vRep
            w := vRep   {the new orbit representative}
        end
        else begin
            CellOrbits^[vRep] := uRep
            w := uRep   {the new orbit representative}
        end
        CellOrbits^[w] := -(uSize + vSize)
    End  {Merge}
```

The function *OrbitRep* follows the pointers *CellOrbits*[v] to the orbit representative.

```
    OrbitRep(v: Integer): Integer
    {follow the pointers to the orbit representative}
    var w: Integer
    Begin
        if CellOrbits^[v] < 0 then return(v)
        w := OrbitRep(CellOrbits^[v])
        CellOrbits^[v] := w   {path compression}
        return(w)
    End  {OrbitRep}
```

These procedures require access to the *CellOrbits* array. It can either be passed as a parameter, or else a global variable can be used to store a pointer to the current *CellOrbits* array.

Comparing Orderings.

As presented in section 3, the procedure *Stabilise* first refines Π, then selects in turn each $u \in C \in \Pi$, creates a partition Π_u in which u is fixed, and calls itself recursively. During the refinement of Π_u, any discrete cells are deleted. The vertices in discrete cells are placed on an array F of fixed points. At least one vertex, u, is moved to the array F during each refinement. The *Refine* procedure will also set the variable *NFixed*, the number of points currently on the array F. In order to have access to this variable, we must alter the calling parameters for *Refine* to include the current searchnode, which contains the partition to be refined.

```
Refine(S: SearchNodePtr)
```

This article appeared in *Computational and Constructive Design Theory*, edited by W.D. Wallis, Kluwer Academic Publishers, 1996.

The first time a discrete partition is obtained, the ordering of F is saved to an array B, the *best* ordering so far. B is a candidate for the canonical ordering of V. The program needs to know whether the ordering B has been intiallized yet. We use a global boolean variable *B_exists* to indicate this condition. It is initially set to *false*. In order to detect whether a refined partition is discrete, the *Refine* procedure can set a global boolean variable *isDiscrete*. The initial search node created containing the unit partition (before refinement) is saved as a global variable, called *TopSearchNode*. The *GroupPtr* that it contains points to Aut(G). When an automorphism is discovered by the program, it is added to the *TopSearchNode*. A refined version of the procedure *Stabilise* is now presented.

```
Stabilise(S: SearchNodePtr)  {refined version}
{S is the current search node, containing a partition Π. Refine Π,
      then select u in the first cell of Π in all inequivalent ways}
Begin
    m := S^.NFixed  {save the number of points currently fixed}
    S^.OnBestPath := false
    Refine(S)  {refine the partition in this search node}
    Result := CompareOrders(m + 1, S^.NFixed)
    if isDiscrete then begin
        if B_exists then begin
            case Result of
            equal: begin  {an automorphism has been found}
                γ := NewPerm(n)  {allocate a new PermPtr}
                for k := 1 to n do γ^[F[k]] := B[k]
                AddGen(γ, TopSearchNode)
                AutoFound := true
            end
            better: begin
                copy F[1..n] to B[1..n]
                set all OnBestPath values to true
            end
            worse: {ignore}
            end  {case}
        end
        else begin
            copy F[1..n] to B[1..n]
```

This article appeared in *Computational and Constructive Design Theory*, edited by W.D. Wallis, Kluwer Academic Publishers, 1996.


```
                B_exists := true
                set all OnBestPath values to true
            end
            goto 1
        end     {if isDiscrete}
        {otherwise the partition is not discrete}
        case Result of
            equal: {ignore}
            better: B_exists := false
            worse: goto 1
        end  {case}
        with S^ do begin
            Π := Partition
            if NextSearchNode=nil then create and initiallize
                NextSearchNode
            S_u := NextSearchNode
        end
        C := first cell of Π
        u := C^.FirstPt
        repeat  {until all inequivalent choices u have been made}
            S_u^.FixedPt := u
            make a copy Π_u of Π in which C is split into {u} and
                C - {u}
            S_u^.Partition := Π_u
            Stabilise(S_u)
            if AutoFound then ...
            dispose(Π_u)
            mark CellOrbits^[u] to indicate that u has been fixed
            change the base of Aut(G) if necessary
            select next inequivalent u ∈ C to stabilise
        until no u was found
1:      {reset degrees to 0 before returning}
        for k := m + 1 to S^.NFixed do Degree^[F[k]] := 0
        S^.FixedPt := 0   {mark this search node inactive}
    End {Stabilise}
```

This article appeared in *Computational and Constructive Design Theory*, edited by W.D. Wallis, Kluwer Academic Publishers, 1996.

The Global Variables.

We collect together here a brief summary of the global variables used by the program, and their purpose.

A: array, the adjacency matrix
V: array, the current ordering of the vertices
B: array, the best ordering of the vertices found so far
F: array, the array of points fixed by refinement
Graph[u]: linked list, the vertices adjacent to u
isDiscrete: Boolean, whether refinement produced a discrete partition
B_exists: Boolean, whether the array B has a value
AutoFound: Boolean, whether an automorphism was found
TopSearchNode: SearchNodePtr, the initial search node
LastBaseChange: SearchNodePtr, see below
BasisOK: Integer, the depth to which the bases agree

When an Automorphism is Discovered.

There are several subtle aspects of the algorithm which are best illustrated by an example. Let G be the graph of Fig. 4. See also the search tree of Fig. 7. The initial unit partition is

$\Pi_0 = \{1, 2, 3, 4, 5, 6\}$

The program will move the points of discrete cells to the array F. However we show the discrete cells included with each partition in this example. The cells of an ordered partition are shown separated by vertical bars.

Stabilise(Π_0) is called, Π_0 is refined
$\Pi_0 = \{1, 2, 3, 4, 5, 6\}$ is equitable
fix 1 :⇒ $\Pi_1 = \{1 \mid 2, 3, 4, 5, 6\}$
  Stabilise(Π_1) is called, Π_1 is refined
  $\Pi_1 = \{1 \mid 6 \mid 2, 3, 4, 5\}$ is equitable, $F = (1, 6)$
  fix 2 :⇒ $\Pi_2 = \{1 \mid 6 \mid 2 \mid 3, 4, 5\}$
    Stabilise(Π_2) is called, Π_2 is refined
    $\Pi_2 = \{1 \mid 6 \mid 2 \mid 4 \mid 3, 5\}$ is equitable, $F = (1, 6, 2, 4)$
    fix 3 :⇒ $\Pi_3 = \{1 \mid 6 \mid 2 \mid 4 \mid 3 \mid 5\}$
      Stabilise(Π_3) is called, Π_3 is refined

This article appeared in *Computational and Constructive Design Theory*, edited by W.D. Wallis, Kluwer Academic Publishers, 1996.

Π_3 is discrete, $B = (1,6,2,4,3,5)$ is assigned
Backtrack to Π_2
fix 5 :⇒ $\Pi_3 = \{1 \mid 6 \mid 2 \mid 4 \mid 5 \mid 3\}$
Stabilise(Π_3) is called, Π_3 is refined
Π_3 is discrete, $F = (1,6,2,4,5,3)$
An automorphism $\gamma = (3,5)$ is discovered.
AddGen is called.
The orbits of each Γ_i on the first cell of Π_i are updated.
The orbits are indicated as follows.
$\Pi_0 = \{(1),(2),(3,5),(4),(6)\}$
$\Pi_1 = \{1 \mid 6 \mid (2),(3,5),(4)\}$
$\Pi_2 = \{1 \mid 6 \mid 2 \mid 4 \mid (3,5)\}$
Backtrack to Π_2
In Π_2 both 3 and 5 have already been fixed
⇒ backtrack to Π_1
In Π_1, only 2 has been fixed so far
fix 3 :⇒ $\Pi_2 = \{1 \mid 6 \mid 3 \mid 2,4,5\}$
Stabilise(Π_2) is called, Π_2 is refined
$\Pi_2 = \{1 \mid 6 \mid 3 \mid 5 \mid 2,4\}$ is equitable, $F = (1,6,3,5)$
fix 2 :⇒ $\Pi_3 = \{1 \mid 6 \mid 3 \mid 5 \mid 2 \mid 4\}$
Stabilise(Π_3) is called, Π_3 is refined
Π_3 is discrete, $F = (1,6,3,5,2,4)$
An automorphism $\gamma = (2,3)(4,5)$ is discovered.
AddGen is called.
The generators of $\mathrm{Aut}(G)$ are now $(3,5)$ and $(2,3)(4,5)$
The orbits of each Γ_i on the first cell of Π_i are updated.
$\Pi_0 = \{(1),(2,3,4,5),(6)\}$
$\Pi_1 = \{1 \mid 6 \mid (2,3,4,5)\}$
$\Pi_2 = \{1 \mid 6 \mid 3 \mid 5 \mid (2,4)\}$
Backtrack to Π_2
In Π_2 points 2 and 4 are in the same orbit
⇒ backtrack to Π_1
In Π_1 points $2,3,4,5$ are in the same orbit ⇒ backtrack to Π_0
fix 2 :⇒ $\Pi_1 = \{2 \mid 1,3,4,5,6\}$
Stabilise(Π_1) is called, Π_1 is refined
$\Pi_1 = \{2 \mid 4 \mid 1,3,5,6\}$ is equitable, $F = (2,4)$

This article appeared in *Computational and Constructive Design Theory*, edited by W.D. Wallis, Kluwer Academic Publishers, 1996.

fix 1 :$\Rightarrow$ $\Pi_2 = \{2 \mid 4 \mid 1 \mid 3,5,6\}$

Stabilise(Π_2) is called, Π_2 is refined

$\Pi_2 = \{2 \mid 4 \mid 1 \mid 6 \mid 3,5\}$ is equitable, $F = (2,4,1,6)$

fix 3 :$\Rightarrow$ $\Pi_3 = \{2 \mid 4 \mid 1 \mid 6 \mid 3 \mid 5\}$

Stabilise(Π_3) is called, Π_3 is refined

Π_3 is discrete, $F = (2,4,1,6,3,5)$

An automorphism $\gamma = (1,2)(4,6)$ is discovered.

AddGen is called.

The generators of $\text{Aut}(G)$ are now

$(3,5), (2,3)(4,5)$, and $(1,2)(4,6)$

The orbits of each Γ_i on the first cell of Π_i are updated.

$\Pi_0 = \{(1,2,3,4,5,6)\}$

$\Pi_1 = \{2 \mid 4 \mid (1,3,5,6)\}$

$\Pi_2 = \{2 \mid 4 \mid 1 \mid 6 \mid (3,5)\}$

Backtrack to Π_2

In Π_2 points 3 and 5 are in the same orbit

$\Rightarrow$ backtrack to Π_1

In Π_1 points $1,3,5,6$ are in the same orbit $\Rightarrow$ backtrack to Π_0

In Π_0 points $1,2,3,4,5,6$ are in the same orbit $\Rightarrow$ done.

The program visited 4 leaf nodes in the search tree, as shown in Fig. 7. Three generators for $\text{Aut}(G)$ were found. It is easy to see that these generators give all of $\text{Aut}(G)$, for the following reason. If the orbits of Γ_i on C_i were not stored, the search algorithm would visit every leaf node of the search tree. Consider the first time an ordering B giving the minimum adjacency matrix was found. The number of leaf nodes with an equivalent ordering is $|\text{Aut}(G)|$. Since every one of these would be visited by the search, every automorphism would be found. By computing the orbits of Γ_i on C_i, the program avoids visiting leaf nodes which it knows to be equivalent to a node already visited. Therefore it will find a representative of every coset of Γ_i in Γ_{i-1}. Consequently the generators found generate all of $\text{Aut}(G)$. The group Γ_0 is initially the identity group, and grows until it equals $\text{Aut}(G)$. Γ_0 is always a subgroup of $\text{Aut}(G)$. So each time a new generator is discovered, Γ_0 must at least double in size. Therefore the number of generators found is at most $\log_2|\text{Aut}(G)|$.

In the example above, all leaf nodes of the search tree are equivalent to each other. Each one gives an automorphism of G. For larger graphs, the

This article appeared in *Computational and Constructive Design Theory*, edited by W.D. Wallis, Kluwer Academic Publishers, 1996.

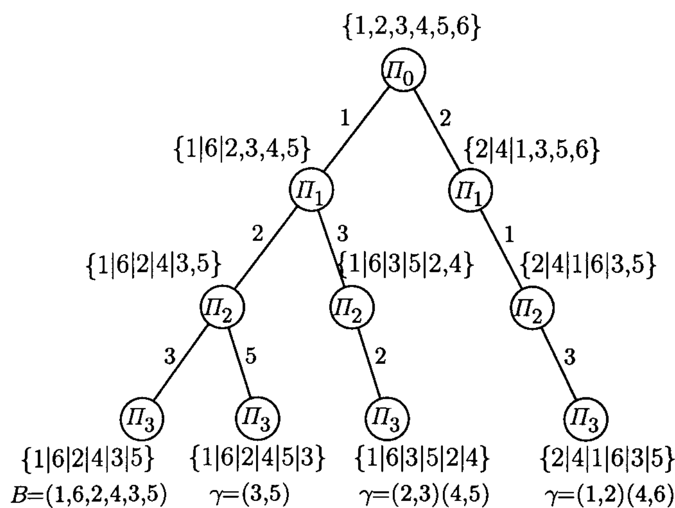

Figure 7 The search tree for the graph of Fig. 6

search tree does not have such a simple structure, even when G is highly symmetric. The ordering B can be set and reset many times. Consider the portion of a search tree shown in Fig. 8. The tree is drawn so that the leaf nodes are visited from left to right.

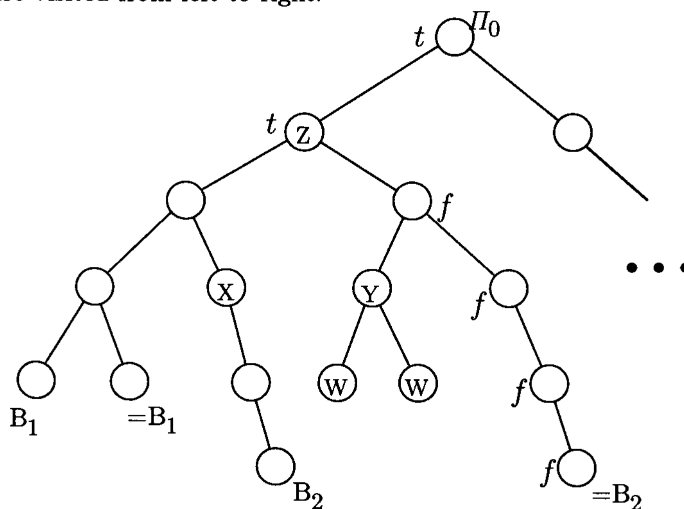

Figure 8 A portion of a search tree

This article appeared in *Computational and Constructive Design Theory*, edited by W.D. Wallis, Kluwer Academic Publishers, 1996.

The first discrete partition occurs at the node labelled B_1. The ordering B is assigned at this point. The next discrete partition gives an equivalent ordering, indicated by $=B_1$ in the diagram. At each level in the recursion the program compares the orderings B and F for the new points fixed at that level. This occurs in the statement "*CompareOrders*($m+1, S$^.NFixed)". It can occur that the new ordering F is already discovered to be better than the previous best ordering B. This is the situation at node X in the diagram. At this point the program sets the flag *B_exists* to *false*. When the next leaf node is reached, the array B is assigned, as if for the first time. This is at node B_2 in the diagram. Notice that the depth of the recursion when a leaf node is discovered can vary. The program then visits the two leaf nodes marked W and finds that the orderings there are worse than the current ordering B_2.

Notice that an automorphism can be missed! These orderings marked W could be equivalent to B_1, but the program does not notice it. In order to avoid this situation, we could store the best two orderings, or the best three, etc. However the more orderings we store, the longer it takes to do a comparison. So far as I know, *Nauty* stores two orderings, the first one found, and the best so far. The current version of *Groups & Graphs* stores only one best ordering.

We make another observation at this point. We have defined a canonical ordering as one which gives a minimum adjacency matrix over all leaf nodes of the search tree constructed by *Stabilise*. It is possible to use the shape of the search tree to further restrict the orderings considered as candidates for the canonical ordering. We could require that only leaf nodes at the minimum possible depth in the recursion are valid candidates. This would exclude the node B_2 in Fig. 8.

The program then proceeds to the node marked $=B_2$ where an ordering equivalent to B_2 is reached. An automorphism is discovered. This means that the current branch of the search tree being searched is equivalent to a branch previously searched. There is no need to continue searching the current branch. How is this detected by the program? The discrete partition at B_2 was obtained by fixing a sequence of vertices $u_0, u_1, u_2, u_3, \ldots$. The partition at $=B_2$ was obtained by fixing another sequence $v_0, v_1, v_2, v_3, \ldots$. These two sequences will usually have some initial vertices in common, in this example, $u_0 = v_0$ but $u_1 \neq v_1$. At node Z, u_1 was first fixed and the branch of the search tree containing B_1 and $=B_1$ were searched. Then v_1 was fixed and the search descended to the branch containing node $=B_2$. The automorphism γ that was discovered maps each u_i to v_i. Therefore

This article appeared in *Computational and Constructive Design Theory*, edited by W.D. Wallis, Kluwer Academic Publishers, 1996.

once γ has been added to Aut(G), the program can ascend up the search tree to node Z, without searching any remaining portion of the current branch. There are several ways to find the node Z. The method used by *Groups & Graphs* works as follows. The path in the search tree from the top to the leaf node containing B is called the best path. The search node data structure contains a boolean value *OnBestPath*. This is initiallized to *false* each time *Stabilise* is entered. Whenever B is assigned, the program executes a loop which starts at *TopSearchNode* and sets the value of *OnBestPath* to *true* for every search node. When the algorithm later descends into other branches of the search tree, some of these will be changed to *false*. This is indicated in Fig. 8 by the letters t and f beside the nodes. The t's were assigned when node B_1 was visited. When an automorphism is discovered, a global flag *AutoFound* is set to *true*. So long as *AutoFound* remains *true* and *OnBestPath* is *false*, the program ascends the search tree. In the example, it will ascend to node Z before another branch is entered. The statement in the program that does this needs to be inserted after the recursive call to *Stabilise*.

```
if AutoFound then begin
    if not S^.OnBestPath then goto 1
    AutoFound := false
end
```

When *Stabilise* is descending the search tree the values *Degree*$[v]$ are not re-initiallized after each refinement. This is because the degree is a cell-invariant of the partition. Before returning to the point where *Stabilise* was called from, it must re-initiallize the degrees, *only* for the vertices which were fixed by the refinement at that level. The reason for this is as follows. At a leaf node, every vertex currently in the partition became fixed. Before returning, the degrees of the vertices just fixed will all be reset to zero. When the calling program in turn is ready to return, it will reset to zero the degrees of the vertices which were fixed at that level, and so on up the tree. This selective re-initiallization saves an enormous amount of execution time.

It is also possible to input to the program known automorphisms of the graph before beginning the search. The group generated by the known automorphisms can prune the search tree significantly. Butler and Lam [8] have developed a very general algorithm that restricts the search to those portions of the search tree known to be equivalent under the action of a specified group.

This article appeared in *Computational and Constructive Design Theory*, edited by W.D. Wallis, Kluwer Academic Publishers, 1996.

Change of Basis.

There is a correspondence between the basis of the automorphism group, and the basis of the partitions created by refinement. Suppose that $u_0, u_1, \ldots, u_{i-1}$ have been fixed to create a partition Π_i. A vertex $u_i \in C_i$ is to be selected for stabilisation. We need to know the orbits of Γ_i on C_i. *Whenever a vertex u is to be selected for stabilisation, the basis of the automorphism group and the basis of the partition must be the same.* Consider again the search tree of Fig. 8. When the automorphism discovered at the node $=B_1$ is added to the group, the basis used for the group will be the current basis of the partition, corresponding to the leftmost path through the search tree. Later, when at node Y vertices are selected for stabilisation, the basis will have changed. This must be reflected in the data structure storing $\mathrm{Aut}(G)$. *The basis of the group is constantly changing.*

As presented in section 4, *AddGen* selects the point u to fix as any point moved by γ. This must be changed so that the basis of the group always agrees with that of the partition.

if S^.FixedPt $\neq 0$ then ItsGroup^.u := S^.FixedPt else select ItsGroup^.u as any point moved by γ

Sometimes the basis of the group will be longer than the basis of the partition. In order to accomodate this possibility, the *FixedPt* of the search node will be set to zero to indicate an inactive search node. This occurs at the end of *Stabilise.*

Before selecting u for stabilisation, the program must check whether the basis of $\mathrm{Aut}(G)$ equals the current sequence of fixed points. The program executes very many base changes. In order to avoid comparing the sequences of stabilised points, we store some additional information. Let $U = u_0, u_1, \ldots$ denote the basis of $\mathrm{Aut}(G)$, and let $U' = u_0', u_1', \ldots$ denote the basis of the partition. The first time an automorphism is discovered, *AddGen* will construct $\Gamma_0, \Gamma_1, \ldots$ with the current basis U', so that the bases agree. When the program acsends the search tree up to a partition Π_i, the bases will still agree up to depth i. If the program now selects a new u_i to stabilise and descends to Π_{i+1} the bases will disagree at this point. The program will continue to descend the tree until a discrete partition is reached. When it then selects another point to stabilise, the bases will still be in agreement up to depth $i-1$. Thus we store a global variable *BasisOK* which contains the depth up to which the bases are in agreement, and a global variable

This article appeared in *Computational and Constructive Design Theory*, edited by W.D. Wallis, Kluwer Academic Publishers, 1996.

LastBaseChange which points to the searchnode whose depth equals *BasisOK*. So when the program is descending the search tree, the value of *BasisOK* doesn't change. When the program is ascending, its value can decrease. We add the following statements right after "*if AutoFound then ...*".

```
if S^.Depth > BasisOK then ChangeBase(S^.Depth)
BasisOK := S^.Depth
LastBaseChange := S
```

The base change algorithm used by *Groups & Graphs* is very straightforward. It simply deletes the generators and coset representatives from the groups at depth greater than *BasisOK* and rebuilds the tower of stabiliser subgroups from this depth. It also re-initiallizes the *CellOrbits*. An alternative way of changing the basis without deleting the existing information is to use the Butler-Sims base change algorithm [8]. It works by converting one basis into another through a sequence of transpositions of consecutive fixed points. My experience with graph isomorphism suggests that it is faster to simply delete the existing data structures and rebuild them. This may not be true for all implementations of the Butler-Sims algorithm.

```
ChangeBase(d: Integer)
{change the basis of the automorphism group, called from depth d}
var    Γ: GroupPtr
       S: SearchNodePtr
Begin
    S := LastBaseChange
    Γ := S^.ItsGroup
    if Γ = nil then return   {no group at this depth of recursion}
    if Γ^.Γ_u = nil then return   {Γ = identity group}
    {first reset the cell orbits from depth BasisOK}
    S := S^.NextSearchNode
    while S ≠ nil do begin
        {search nodes below depth d do not correspond to partitions/}
        if S^.Depth ≤ d then begin
            C := first cell of S^.Partition
            for i := C^.FirstPt to C^.LastPt do
                S^.CellOrbits^[V[i]] := −1
        end
```

This article appeared in *Computational and Constructive Design Theory*, edited by W.D. Wallis, Kluwer Academic Publishers, 1996.


```
            S := S^.NextSearchNode
        end
        save the generators from Γ
        delete coset representatives from Γ
        delete generators and coset representatives for all groups
            in Γ_u tower
        for each generator γ of Γ do AddGen(γ, LastBaseChange)
    End {ChangeBase}
```

Marking the Cell Orbits.

When a point $u_i \in C_i$ has been stabilised in the search node containing Π_i, the orbit of Γ_i containing u_i must be marked to indicate that it is not necessary to stabilise any more points from that orbit. If this orbit later merges with other orbits when an automorphism is discovered, the resulting orbit must also be marked to indicate that a point from it has already been fixed. One way to do this is to utilise the *CellOrbits* array. Currently the value stored in *CellOrbits*[v] has the following property.

$$CellOrbits[v] > 0, \text{ if } v \text{ is not the representative of Orb}(v)$$
$$CellOrbits[v] = -|Orb(v)|, \text{ if } v \text{ is the representative of Orb}(v)$$

If v is an orbit representative, we can use the value stored in *CellOrbits*[v] to mark an orbit. If n is the (global) number of points in the graph G, we can set

$$CellOrbits[v] = -|Orb(v)|, \text{ if no point from Orb}(v) \text{ has been fixed}$$
$$CellOrbits[v] = -|Orb(v)| - n, \text{ if a point from Orb}(v) \text{ has been fixed}$$

The only effect this has is to require a slight modification to the procedure which merges orbits when an automorphism is added to the group. It also has the advantage of not requiring another array to mark the orbits, since another array would have to be re-initiallized every time the basis is changed.

This article appeared in *Computational and Constructive Design Theory*, edited by W.D. Wallis, Kluwer Academic Publishers, 1996.

6 AFTERWORD

Many modifications to these ideas and methods are possible. It is my experience that there are many people who are interested in writing isomorphism programs, but who are somewhat daunted by the difficulty and the lack of information on specific programming techniques. I have endeavored to provide the results of my experience programming graph isomorphism, in order to highlight the main ideas involved and to point out some of the difficulties, subtleties, and unexplored possibilities.

REFERENCES

[1] A. Aho, J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison Wesley, Toronto, 1974.

[2] L. Babai, P. Erdös, S.M. Selkow, Random graph isomorphism, *SIAM J. of Computing* 9 (1980), 628–635.

[3] Laszlo Babai and Eugene Luks, Canonical labeling of graphs,*Proc. 15th Annual ACM Symp. on Theory of Computing*, SIGACT, 1983.

[4] J.M. Bennett and J.J. Edwards, A graph isomorphism algorithm using pseudoinverses, preprint, Computer Science Department, University of Sydney.

[5] N.L. Biggs and A.T. White, *Permutation Groups and Combinatorial Structures*, London Math. Soc. Lecture Notes #33, Cambridge Univ. Press, Cambridge, 1979.

[6] Bela Bollobas,*Graph Theory, an Introductory Course*, Springer-Verlag, New York, 1979.

[7] Gregory Butler and John Cannon, Computing in permutation and matrix groups I: normal closure, commutator subgroups, series, *Mathematics of Computation* 39 (1982), 663–670.

[8] G. Butler and C.W.H. Lam, A general backtrack algorithm for the isomorphism problem of combinatorial objects, *J. Symbolic Computation* 1 (1985), 363–381.

[9] Derek Corneil and Mark Goldberg, A non-factorial algorithm for canonical numbering of a graph, Technical Report #160/82, 1982, Department of Computer Science, University of Toronto.

This article appeared in *Computational and Constructive Design Theory*, edited by W.D. Wallis, Kluwer Academic Publishers, 1996.

[10] D.G. Corneil and C.C. Gotlieb, An efficient algorithm for graph isomorphism, *JACM* 17 (1970) 51–64.

[11] D.G. Corneil and D.G. Kirkpatrick, A theoretical analysis of various heuristics for the graph isomorphism problem, *SIAM J. Computing* 9 (1982), 281–297.

[12] M. Furst, J.E. Hopcroft, and E. Luks, A subexponential algorithm for trivalent graph isomorphism, 1980, Tech. Rept. 80–426, Dept. of Computer Science, Cornell University.

[13] R.M. Karp, Probabilisitic analyisis of a canonical numberiung algorithm for graphs, *Proc. Symp. Pure Math.* 34 (1979), 365–378.

[14] Zvi Galil, Christoff M. Hoffmann, Eugene M. Luks, Claus P. Schnorr, and Andreas Weber, An $O(n^3 \log n)$ deterministic and $O(n^3)$ probabilistic isomorphism test for trivalent graphs, *Proc. 23rd IEEE Symp. on the Foundations of Comp. Sci.*, N.Y., 1982, pp. 118–125.

[15] S.J. Gismondi and E.R. Swart, A polynomial-time procedure for resolving the graph isomorphism problem, Department of Computing and Information Science, UIniversity of Guelph, Ontario, preprint.

[16] Christoff Hoffmann, *Group Theoretic Algorithms and Graph Isomorphism*, Lecture Notes in Computer Science #136, Springer-Verlag, New York, 1982.

[17] Andrew Kirk, Efficiency considerations in the canonical labelling of graphs, Technical report TR-CS-85-05, Computer Science Department, Australian National University (1985).

[18] William Kocay, Groups & Graphs, a Macintosh application for graph theory, *J. Combinatorial Mathematics and Combinatorial Computing* 3 (1988), 195–206.

[19] J.S. Leon, An algorithm for computing the automorphism group of a Hadamard matrix, *J. Combinatorial Theory* 27A (1979) 289–306.

[20] Eugene M. Luks, Isomorphism of graphs of bounded valence can be tested in polynomial time, *Proc. 21st IEEE Symp. on the Foundations of Comp. Sci.*, Rochester, N.Y., 1980, 42–49.

[21] Rudi Mathon, Sample graphs for isomorphism testing, *Conressus Numerantium* 21 (1978) 499–517.

This article appeared in *Computational and Constructive Design Theory*, edited by W.D. Wallis, Kluwer Academic Publishers, 1996.

[22] Brendan McKay, *Topics in Computational Graph Theory*, Ph.D. Thesis, University of Melbourne, 1980.

[23] Brendan McKay, *Nauty User's Guide* (Version 1.5), Computer Science Department, Australian National University.

[24] Brendan McKay, Practical graph isomorphism, *Conressus Numerantium* 30 (1981), 45–87.

[25] Brendan McKay, *Backtrack Programming and the Graph Isomorphism Problem*, M.Sc. Thesis, University of Melbourne, 1976.

[26] R.C. Read and D.G. Corneil, The graph isomorphism disease, *J. Graph Theory* 1 (1977), 339–363.

[27] Mark Allen Weiss, *Data Structures and Algorithm Analysis*, Benjamin Cummings, Redwood City, California, 1992.

[28] Helmut Wielandt, *Finite Permutation Groups*, Academic Press, New York, 1964.