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]. 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.

1Nauty is available via anonymous ftp from dcssoft.anu.edu.au, in /pub/nauty19.
2Groups & Graphs is available at ftp.cc.umanitoba.ca in the directory /pub/mac.

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 (e.g., 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. Conneil and Goldberg have presented a non-factorial algorithm 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.
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 $u \rightarrow v$ to indicate that $u$ is adjacent to $v$. Since $G$ is undirected we also have $v \rightarrow u$. This can also be expressed as $uv \in E(G)$.

The graph isomorphism program we are describing produces a certificate for $G$, written $\text{cert}(G)$. Two graphs $G$ and $H$ are isomorphic if and only if they have equal certificates, $\text{cert}(G) = \text{cert}(H)$ (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 11001001101001100111001001. 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 $\text{cert}(G)$. We say that $\text{cert}(G)$ corresponds to the smallest adjacency matrix for $G$. When defined in this way, $\text{cert}(G)$ 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 $\Pi$ 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, $\bigcup C_i = V(G)$, and $C_i \cap C_j = \emptyset$, for $i \neq j$. The degree sequence of $G$ defines an ordered partition of $G$. A partition $\Pi$ 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 $\Pi$ is not stable, then we can refine it, as defined by
the following algorithm. We assume that there is a global array $\text{Degree}[:,]$, such that $\text{Degree}[v]$ is a cell invariant for $\Pi$. Initially we can assume that $\text{Degree}[v] = 0$, for all $v = 1, 2, \ldots, n$.

\textbf{Refine}(\Pi; \text{partition})

\{ $\Pi$ is an ordered list of cells \}
\{ each cell of $\Pi$ is initially marked uncounted \}

Begin

\begin{itemize}
  \item[$C$] := first cell of $\Pi$
  \Repeat
    \{ first count the adjacencies to cell $C$ \}
    \For each $u \in C$
      \For each $v \rightarrow u$
        \text{Degree}[v] := $\text{Degree}[v] + 1$
      \EndFor
    \EndFor
    \Mark $C$ \textit{counted}
    \For each cell $C'$ do begin
      \Sort the vertices of $C'$ by $\text{Degree}[:,]$
      \If $\text{Degree}[:,]$ is not constant on $C'$ then
        \Split $C'$ into new cells of equal degree
        \All new cells created are marked \textit{uncounted}
      \EndIf
    \EndFor
    $C$ := an uncounted cell of $\Pi$
  \Until $\Pi$ is stable
\End {Refine}
On Writing Isomorphism Programs

For example, let us start with the unit partition $\Pi_0$ 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 $\text{Degree}[v] = 3$ and every $v \in C_2$ has $\text{Degree}[v] = 4$. In order to determine that $\Pi_1$ 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 $\text{Degree}[v]$ 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 $\Pi_1$ 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 $\Pi$. 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. $\Pi$ is smaller than $\Pi'$ if it is finer than $\Pi'$. 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 $\Pi$ and $\Pi'$, there is a unique partition $\Pi \wedge \Pi'$ which is the largest partition smaller than both $\Pi$ and $\Pi'$. It is obtained by forming all pairwise intersections of the cells of $\Pi$ and $\Pi'$. There is also a unique partition $\Pi \vee \Pi'$ which is the smallest partition larger than both $\Pi$ and $\Pi'$. It is obtained by taking the unions of intersecting cells of $\Pi$ and $\Pi'$. See Fig. 2. If we formed a bipartite graph of the cells of $\Pi$ versus the cells of $\Pi'$, and joined intersecting cells, then the cells of $\Pi \vee \Pi'$ would correspond to the connected components.
of this graph. The cells of $\Pi \land \Pi'$ would correspond to the edges of this bipartite graph.

![Diagram](image)

**Figure 2** Partitions form an ordered set

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

This can be proved by counting adjacencies using the diagram above. Note however, that $\Pi \land \Pi'$ is not always equitable. See McKay [25]. Now consider the refinement procedure in which a non-equitable partition $\Pi$ is to be refined. A cell $C$ of $\Pi$ is selected and counted. This causes some of the cells of $\Pi$ to split into two or more cells. Let $\Pi_C$ 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 $\Pi_C \lor \Pi_{C'}$ would be an equitable partition. Since $\Pi_C \land \Pi \leq \Pi$ and $\Pi_{C'} \land \Pi \leq \Pi$, it follows that $\Pi_C \lor \Pi_{C'} \leq \Pi$. But this is impossible. Thus for any unordered partition $\Pi$ there is a unique largest equitable unordered partition $\Pi'$ that is smaller than $\Pi$. There are a number of equitable ordered partitions that correspond to $\Pi'$. 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 $\Pi'$ that it constructs is unique.

The termination of the loop "repeat ... until $\Pi$ 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 $\Pi$. 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 $\Pi$ is equitable. So one way to recognize an equitable partition is to continue until every cell has been
On Writing Isomorphism Programs

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 \( \Pi \) can become discrete at such a step.

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

1. if there is no uncounted cell \( C \), then \( \Pi \) is stable;
2. if \( \Pi \) 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 \( \text{Degree}[v] \), 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 \( \Pi \) 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 \( u \), \( \text{Graph}[u] \) denotes the list of vertices adjacent to \( u \). This allows the loop "for all \( v \rightarrow u \) do" to be programmed efficiently. An adjacency matrix is also stored, since it is needed for comparing different orderings of \( V(G) \) when finding \( \text{cert}(G) \). 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.

```plaintext
CellPtr = ^Cell
Cell = record
  FirstPt, LastPt: Integer
  Counted, Discrete: Boolean
  NextCell: CellPtr
end
```
FirstPt and LastPt are pointers into the array $P[i]$. The vertices in each cell are stored as contiguous entries in the array. Counted indicates whether the cell has been counted, and Discrete 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 $v$ 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($\Pi$) accepts an ordered partition $\Pi$ and transforms it into an equitable ordered partition $\Pi'$. Viewed as unordered partitions, $\Pi'$ is the largest equitable partition which is less than or equal to $\Pi$. The program assumes that initially Degree[$v$] is a cell-invariant of $\Pi$ (Degree[$v$] = 0 the first time Refine is called). When it terminates, $\Pi$ has been transformed into an equitable partition $\Pi'$, so that Degree[$v$] will still be a cell-invariant of $\Pi'$. This feature is significant, because it means that the program does not need to re-initialize 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 $\Pi$ 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.
On Writing Isomorphism Programs

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 \(e\) edges, the number of steps needed to count a cell \(C\) is at most \(O(e)\). If \(\Pi\) is already equitable, the total number of steps required to detect this is \(2e\), by counting every cell. Since there are at most \(n\) splittings, the total number of steps used in counting cells is at most \(O(ne)\). 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 \(\text{Aut}(G)\). If \(\pi, \tau \in \text{Aut}(G)\), the image of vertex \(v\) under \(\pi\) is denoted \(v^\pi\). Permutations are composed from left to right, so that \(v^\pi\tau\) indicates the image of \(v\) under the product \(\pi \tau\) (first \(\pi\), then \(\tau\)). A cell \(C\) of a partition \(\Pi\) is fixed by \(\text{Aut}(G)\) if \(C^\pi = C\), for every \(\pi \in \text{Aut}(G)\), that is, \(\pi\) maps every element...
of $C$ to an element of $C$. If $\Pi = (C_1, C_2, \ldots, C_p)$, then $\Pi^*$ denotes the ordered partition whose cells are $(C_1^*, C_2^*, \ldots, C_p^*)$. The connection between partition refinement and the automorphism group is the following.

**Lemma 2.2** Let $\Pi_0$ be a partition of $V(G)$ such that $\text{Aut}(G)$ fixes every cell of $\Pi_0$. Let $\Pi$ be obtained from $\Pi_0$ by refinement. Then $\text{Aut}(G)$ fixes every cell of $\Pi$.

**Proof.** The proof is by induction on the number of iterations of the repeat loop. Initially all automorphisms fix every cell of $\Pi_0$. 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 $\pi \in \text{Aut}(G)$ maps $u \in C$ to $v \in C'$, then $\pi$ also maps the vertices adjacent to $u$ to the vertices adjacent to $v$. So $\pi$ fixes every new cell split from $C'$. It follows that $\pi$ fixes every cell of $\Pi$.

Since the unit partition is always fixed by $\text{Aut}(G)$, for every graph $G$, we can always take $\Pi_0$ to be the unit partition initially, and refine it to an equitable partition $\Pi$. 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 $\Pi$, 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 $\Pi$-ordering of $G$ is any permutation of $V(G)$ that fixes every cell of $\Pi$ setwise. The number of $\Pi$-orderings is $\prod_{i=1}^{p} |C_i|!$. We could then define $\text{cert}(G)$ as the smallest adjacency matrix over the set of $\Pi$-orderings of $V(G)$. For example, if we compute $\text{Refine}(\Pi_0)$ for the graph of Fig. 3, we find that the resulting equitable partition is discrete. This uniquely defines the certificate for $G$.

![Figure 3](image-url) A graph with discrete partition
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 $\Pi$ be a non-discrete equitable partition of $G$. We can assume that $\Pi$ has no discrete cells. Let $\Gamma$ be a subgroup of $\text{Aut}(G)$ such that $\Gamma$ fixes $\Pi$. If $C$ is any cell of $\Pi$, and $u \in C$ is any vertex of $C$, then in some of the $\Pi$-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 $\Pi'_u$. $\Gamma$ no longer fixes $\Pi'_u$, 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 $\Pi'_u$.

3.1 Let $\Gamma$ be a permutation group acting on a set $V$. The subgroup $\Gamma_u$ consisting of those permutations that fix $u$ is called a stabiliser subgroup.

We now refine $\Pi'_u$ to an equitable partition $\Pi_u$. By Lemma 2.2, $\Gamma_u$ fixes $\Pi_u$. This process of fixing a vertex $u$ in a partition $\Pi$ and refining $\Pi'_u$ to an equitable partition $\Pi_u$ is called vertex stabilisation. In most graphs it produces a considerable refinement of $\Pi$.

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

The number of $\Pi_u$-orderings of $V$ will be much less than the number of $\Pi$-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(\Pi; partition) \{first version\}
{refine $\Pi$ 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(\Pi)
if $\Pi$ is discrete then begin
II defines an ordering of the vertices
compare it with the best ordering found so far,
replacing the best if necessary
return
{otherwise II is not discrete}
\( C := \text{first cell of II} \)
for each \( u \in C \) do begin
make a copy \( \Pi_u \) of \( \Pi \) in which \( C \) is split into
\( \{u\} \) and \( C - \{u\} \)
\( \text{Stabilise}(\Pi_u) \)
\( \text{Dispose}(\Pi_u) \)
\{since some vertex of \( C \) must be first, and we have tested
all vertices of \( C \), at this point we have \( \text{cert}(G) \)\}
end
End  \{Stabilise\}

The calling program begins by setting up a unit partition of \( V \) and calling \( \text{Stabilise}(\Pi) \). 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 \( \text{Refine}(\Pi) \) is executed, one or more discrete cells may be produced. Once a discrete cell has been counted, we remove it from \( \Pi \), 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 \( N\text{Fixed} \). 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...
On Writing Isomorphism Programs

147

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

Function \texttt{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
\quad if \(i = 1\) then \(i := 2\) \{nothing to compare in column 1\}
\quad for \(c := i\) to \(j\) do \{column \(c\}\}
\quad \quad for \(r := 1\) to \(c - 1\) do begin \{row \(r\}\}
\quad \quad \quad if \(A[F[r], F[c]] < A[B[r], B[c]]\) then return(1)
\quad \quad \quad if \(A[F[r], F[c]] > A[B[r], B[c]]\) then return(-1)
\quad \quad end
\quad return(0)
end \{Compare\}

We now define \(\text{cert}(G)\) as the smallest adjacency matrix with respect to the orderings produced by \(\text{Stabilise}(II)\). 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, \ldots, \(n - 1\) 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 \(\pi\) is a permutation of \(V\), \(A^\pi\) denotes the adjacency matrix obtained from \(A\) by permuting the rows and columns by \(\pi\). Two orderings \(\pi_1\) and \(\pi_2\) 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 \(\pi_1\pi_2^{-1} \in \text{Aut}(G)\).

Lemma 3.1 The number of inequivalent orderings of \(V\) is \(n!/[\text{Aut}(G)]\).
Proof. Two orderings $\pi_1$ and $\pi_2$ are equivalent if and only if $\pi_1 \pi_2^{-1} \in \text{Aut}(G)$. So each coset of $\text{Aut}(G)$ in the symmetric group $\text{Sym}(V)$ 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 $\Gamma$ is a permutation group acting on $V$, $\pi$ is any element of $\Gamma$, $u$ is any element of $V$ and $v = u^\pi$. The order of a group $\Gamma$ 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 $\Gamma$ that map $u$ to $v = u^\pi$.

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

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

3.7 The conjugate of the subgroup $\Gamma_u$ is $\pi^{-1} \Gamma_u \pi = \Gamma_u^\pi = \Gamma_u$, that is, the conjugate of the stabiliser of $u$ by $\pi$ is the stabiliser of $v = u^\pi$.
3.8 If $\Gamma_u$ fixes a partition $\Pi$, then $\Gamma_u^*$ fixes $\Pi^*$.

Lemma 3.2 Suppose that $\Gamma \leq \text{Aut}(G)$ fixes $\Pi$, and let $\Pi_u$ be obtained by partition refinement. If $\pi$ is any element of $\Gamma$, let $\nu = \pi^*$. Then $\Pi_\nu = \Pi_u^*$.

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

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. $\Pi_0$ is the initial equitable partition obtained by refining the unit partition. A vertex $u_0$ in $\Pi_0$ is fixed, and after refinement, $\Pi_1$ is obtained. Then $u_1$ in $\Pi_1$ is fixed, and so on, until the discrete partition $\Pi_k$ is reached. We assume throughout that discrete cells are deleted as they are encountered, so that $\Pi_0, \Pi_1, \ldots, \Pi_k$ do not have any discrete cells. Only $\Pi_k$ has discrete cells. Vertices $u_0, u_1, \ldots, u_{k-1}$ have all been fixed in succession. As outlined above, the program $\text{Stabilise}(\Pi)$ always selects $u$ in the first cell $C$ of $\Pi$. 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 $\Pi_i$. Then $u_i \in C_i$, for $i = 0, 1, \ldots, k-1$. Let $\pi \in \text{Aut}(G)$, and let $v_i = u_i^*$, for $i = 0, \ldots, k-1$. Then $\Pi_0^* = \Pi_0$. Fixing $v_0$ instead of $u_0$ gives the partition $\Pi_1^*$, by Lemma 3.10. Since $u_1 \in C_1$, we know that $v_1 \in C_1^*$, the first cell of $\Pi_1^*$. Fixing $v_1$ in $\Pi_1^*$ gives the partition $\Pi_2^*$, and so on. Thus, fixing $u_0^*, u_1^*, \ldots, u_{k-1}^*$ gives a sequence $\Pi_0^*, \Pi_1^*, \ldots, \Pi_k^*$ of partitions, such that the discrete partition $\Pi_k^*$ is equivalent to $\Pi_k$.

A sequence $u_0, u_1, \ldots, u_{k-1}$ of points fixed in order to produce a discrete partition $\Pi_k$ is termed a basis for the ordering of $V$ defined by $\Pi_k$. The search tree constructed by $\text{Stabilise}$ 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 $(u_0, \ldots, u_j, u_{j+1})$ descends
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 \(\Pi_{j+1}\) constructed by fixing these points in succession. The root node corresponds to \(\Pi_0\). 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 \(\pi\) induces an automorphism of the search tree. We know that \(\text{Aut}(G)\) fixes \(\Pi_0\). The algorithm selects \(u_0\) in \(\Pi_0\) and fixes it to get \(\Pi_1\). The branch of the search tree descending from \(\Pi_0\) on the edge labelled \(u_0\) will be searched by \(\text{Stabilise}(\Pi_1)\). If \(u_0^\pi\) in \(\Pi_0\) is fixed instead, to get \(\Pi_1^\pi\), then the branch descending on the edge labelled \(u_0^\pi\) will be isomorphic to the branch just searched, by Lemma 3.10. So having fixed \(u_0\), we need fix no other points \(u_0^\pi\) for any \(\pi \in \text{Aut}(G)\). Write \(\Gamma_0 = \text{Aut}(G)\). Let \(\Gamma_1\) denote the stabiliser of \(u_0\) in \(\Gamma_0\). Then \(\Gamma_1\) fixes \(\Pi_1\). In general, let \(\Gamma_i\) denote the subgroup of \(\Gamma_{i-1}\) obtained by stabilising \(u_{i-1}\). Then \(\Gamma_i\) fixes \(\Pi_i\). The algorithm selects some \(u_i \in \Pi_i\) and fixes it. Let \(\pi \in \Gamma_i\). The branch of the search tree descending from \(\Pi_i\) on the edge labelled \(u_i\) will be isomorphic to the branch descending on the edge labelled \(u_i^\pi\). So having chosen \(u_i \in \Pi_i\), we need choose no other points \(u_i^\pi\) for any \(\pi \in \Gamma_i\). That is, we need to fix exactly one point from each orbit of \(\Gamma_i\) on \(C_i\) in order to find the unique minimum adjacency matrix of \(G\).

![Diagram](image.png)

**Figure 4** The graph of the octahedron

Initially the algorithm does not know \(\text{Aut}(G)\) or any of the stabiliser subgroups \(\Gamma_i\). However as the program executes it will occasionally find a leaf node whose ordering of \(V\) is equivalent to the best ordering. By Lemma
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 $\pi$ 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 $\pi$ is a permutation that maps $u$ to $v = u^\pi$, then there is an edge from $u$ to $v$ labelled $\pi$. 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.

\[ V = \{1, 2, 3, 4, 5, 6\} \]
\[ \pi = (1, 6, 2, 4, 5, 3) \]
\[ \tau = (1)(2, 5)(3, 6)(4) \]

Figure 5 A permutation diagram

Let $\Gamma$ denote the group generated by $\pi$ and $\tau$. Suppose that we are given $\pi$ and $\tau$ and want to find $\Gamma$ and a sequence of stabiliser subgroups. $\Gamma$ consists of all products that can be formed from $\pi$, $\tau$ and their inverses. It is very easy to construct the permutation diagram for $\pi$ and $\tau$. Notice that each connected component of the permutation diagram represents an orbit of $\Gamma$. 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 $\pi\tau\pi\tau\pi^{-1}$. We come to vertices 1, 6, 2, 5, 3, 6, 1, in that order. We conclude that the product $\pi\tau\pi\tau\pi^{-1}$ maps 1 to 1, that is, it is an element of the stabiliser of 1 in $\Gamma$. Thus we have the following important observation.
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 \( \Gamma_u \) 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 \( \Gamma \) mapping \( u \) to \( v \), that is, a right coset of \( \Gamma_u \) (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 \( \Gamma \) 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 \( \Gamma \) into right cosets of \( \Gamma_u \), where \( u \) is any point in \( X \).

The spanning tree in Fig. 5 is rooted at point 5. The cosets of \( \Gamma_5 \), 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 \( \Gamma_u \), then together with the coset representatives, this would completely determine \( \Gamma \).

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 \( \pi \). 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 \( \tau \) is used, the fundamental cycle is \((3,1,6)\) and the walk \( T_6 \pi T_3^{-1} \) gives the sequence \((5,3,1,6,3,5)\) of vertices, which corresponds to the product \( \pi \pi \pi \pi^{-1} \) of the generators. Any edge \( u \) 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 Bollobás [6] for a proof. We
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 $v_1 w_1, v_2 w_2, \ldots, v_k w_k$ not in $T$. If $k = 1$ then $C$ is the fundamental cycle $C_{v_1 w_1}$. $T_{v_1} v_1 w_1 T_{w_1}^{-1}$ 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 $T_{v_1} v_1 w_1 T_{w_1}^{-1} \ldots T_{v_k} v_k w_k T_{w_k}^{-1}$ 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 $\Gamma_u$, 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 $\Gamma_u$ is generated by the products of the generators given by the walks $T_u w T_w^{-1}$, 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 $\Gamma$ 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.
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 $\Gamma$ contains a linked list of generators, an orbit $\text{Orb}(u)$, for some point $u \in V$, and a representative for each coset of $\Gamma_u$. 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 $\Gamma_u$. Since $\Gamma_u$ 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 $\text{Orb}(u)$ \}
  Cosets: CosetPtr  \{ coset reps of $\Gamma_u$ \}
  Inverses: CosetPtr
  $\Gamma_u$: GroupPtr  \{ stabiliser of $u$ \}
end

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

**GroupElt**(\gamma: PermPtr; \Gamma: GroupPtr): Boolean
\{returns true if $\gamma \in \Gamma$\}
var $\pi$: PermPtr
Begin
On Writing Isomorphism Programs

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^[u] = nil then return(false) {v ∈ Orb(u)}
π := MultiplyPerm(γ, Inverses^[v])
{π maps u to u}
return(GroupId(Γ, π))
end
End {GroupId}

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 = uγ is found, and the program checks whether v ∈ Orb(u). If not, then γ /∈ Γ. Otherwise let γ_v denote the coset representative stored for the point v (γ_v = Cosets^[v]). The product π = γγ_v⁻¹ is computed. Clearly π fixes u, so that γ ∈ Γ if and only if π ∈ Γ_u. Therefore the program makes a recursive call. A function MultiplyPerm is needed in order to compute the product γγ_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
subgroup $\Gamma_u$. The chords of the spanning tree are used to construct generators for $\Gamma_u$. The main procedure \texttt{AddGen} takes an existing group $\Gamma$ which is already stored. The first time it is called, $\Gamma$ will have been initialized to the identity group on $n$ points. This is a \texttt{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 \texttt{Cosets} and \texttt{Inverses} are allocated and every entry is initialized to \texttt{nil}, and the stabiliser subgroup $\Gamma_u$ is set to \texttt{nil}. The program takes a permutation $\gamma$ which is a generator of $\Gamma$, but not yet represented in the data structure. Thus, $\gamma \not\in \Gamma$ when the procedure is called. It updates the data structure so that upon completion $\gamma$ is now recognized as a generator of $\Gamma$.

\begin{verbatim}
AddGen($\gamma$: PermPtr; $\Gamma$: GroupPtr)
    \{add $\gamma$ to the data structure representing $\Gamma$\}
    var $\pi$: PermPtr
    Begin with $\Gamma^u$ do begin
        if $\Gamma_u = \text{nil}$ then begin
            \{initialize $\Gamma_u$ to the identity group on $n$ points\}
            $\Gamma_u := \text{IdentityGroup}(n)$
            \{find a point $u$ moved by $\gamma$\}
            $u := 1$
            while $\gamma^u = u$ do $u := u + 1$
        end
        add $\gamma$ to the linked list \texttt{Generators}
        $M := \text{NPts}$ \{current number of points in Orb($u$)\}
        $k := 1$
        while $k \leq M$ do begin
            $v := \text{Orbit}^u[k]$ \{$k^{th}$ point in the orbit\}
            $w := \gamma^v$
            if $\text{Cosets}^v[w] = \text{nil}$ then begin
                \{$w \not\in \text{Orb}(u)$\}
                $\text{NPts} := \text{NPts} + 1$
                $\text{Orbit}^{\text{NPts}} := w$ \{add $w$ to \text{Orb}($u$)\}
                $\text{Cosets}^v[w] := \text{MultiplyPerm}(\text{Cosets}^v[w], \gamma)$
            end
            else begin
                \{$w \in \text{Orb}(u)$\}
                if $\text{Inverses}^v[w] = \text{nil}$ then
                    $\text{Inverses}^v[w] := \text{InversePerm}(\text{Cosets}^v[w])$
                    $\pi := \text{GenMultiply}(\text{Cosets}^v[w], \gamma, \text{Inverses}^v[w])$
                end
            end
        end
    end
\end{verbatim}
On Writing Isomorphism Programs

157

if not GroupElt(\(\pi\), \(\Gamma_u\)) then AddGen(\(\pi\), \(\Gamma_u\))
else Dispose(\(\pi\)) \{\(\pi\) is not needed\}
end

\(k := k + 1\)
end

\{if \(\gamma\) has extended Orb(\(u\)), apply all generators to all new points\}
while \(k \leq \text{NPts}\) do begin

\(v := \text{Orbit}^k[u]\) \{\(k\)th point in the orbit\}
for each generator \(\tau \in \text{Generators}\) do begin

\(w := \tau^v[u]\)
if Cosets\(\gamma\)^{\omega}[w] = \text{nil} then begin \{\(w \not\in \text{Orb}(u)\}\}

\(\text{NPts} := \text{NPts} + 1\)
\(\text{Orbit}^{\text{NPts}} := w\) \{add \(w\) to \(\text{Orb}(u)\)\}
\(\text{Cosets}^{\omega}[w] := \text{MultiplyPerm}(\text{Cosets}^{\omega}[w], \tau)\)
end
else begin \{\(w \in \text{Orb}(u)\}\}

if Inverses\(\gamma\)^{\omega}[w] = \text{nil} then
\(\text{Inverses}^{\omega}[w] := \text{InversePerm}(\text{Cosets}^{\omega}[w])\)
\(\pi := \text{GenMultiply}(\text{Cosets}^{\omega}[w], \tau, \text{Inverses}^{\omega}[w])\)
if not GroupElt(\(\pi\), \(\Gamma_u\)) then AddGen(\(\pi\), \(\Gamma_u\))
else Dispose(\(\pi\)) \{\(\pi\) is not needed\}
end

\(k := k + 1\)
end end \{AddGen\}

The procedure first computes \(w = v^\gamma\) for all \(v \in \text{Orb}(u)\). Let \(\gamma_v = \text{Cosets}^{\omega}[v]\) be the coset representative for \(v\). If \(w \not\in \text{Orb}(u)\), then \(w\) is added to the orbit. \(\gamma_v^\gamma\) maps \(u\) to \(w\), so it becomes \(\gamma_w\), the coset representative for \(w\). It is computed by \(\text{MultiplyPerm}\). 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 \(\Gamma_u\). \text{GenMultiply} is a procedure that performs this multiplication. This is more efficient than calling \text{MultiplyPerm} twice.

\text{GenMultiply}(\(\alpha, \beta, \gamma\): PermPtr): PermPtr
\{compute \(\alpha \beta \gamma\}\)
var \(\pi\): PermPtr
begin
    \( \tau := \text{NewPerm}(n) \) \{ allocate a new perm on \( n \) points \}
    for \( k := 1 \) to \( n \) do \( \tau^\wedge[k] := \gamma^\wedge[\beta^\wedge[\alpha^\wedge[k]]] \}
    return(\( \tau \))
end \{GenMultiply\}

The program calls GroupElt to check if \( \pi \) is known to be in \( \Gamma_u \). If so it is discarded. The storage used by it is reclaimed by Dispose, an operating system call. If \( \pi \) is not currently known to be in \( \Gamma_u \) then AddGen is called recursively. When it returns, the data structure representing \( \Gamma_u \) will have been modified to account for \( \pi \). Once \( \gamma \) has been applied to all \( v \in \text{Orb}(u) \), the program applies all generators of \( \Gamma \) 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 \( v \in \text{Orb}(u) \), and the stabiliser \( \Gamma_u \) is up to date. Therefore the data structure for \( \Gamma \) is also up to date.

The sequence of points fixed in order to construct the tower of stabilisers is called the \textit{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 \( \Gamma \). 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.
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 $\Gamma$ 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 $\Pi_i$ was stabilised in order to obtain $\Pi_{i+1}$. A sequence of groups $\Gamma_0, \Gamma_1, \ldots, \Gamma_k$ is associated with the partitions. $\Gamma_0 = \text{Aut}(G)$. $\Gamma_{i+1}$ is the stabiliser of $u_i$ in $\Gamma_i$, where $i = 0, 1, \ldots, k - 1$. $\Gamma_i$ fixes $\Pi_i$. $\Gamma_k$ is the identity group. So the basis $u_0, u_1, \ldots, u_{k-1}$ of the discrete partition $\Pi_k$ is also a basis for the group $\Gamma_0$. Associated with each node of the search tree is a partition $\Pi_i$ and a group $\Gamma_i$. Associated with each descent from $\Pi_i$ to $\Pi_{i+1}$ is the vertex $u_i$. $u_i$ is contained in the first cell $C_i$ of $\Pi_i$. In order to select a vertex from each orbit of $\Gamma_i$ acting on $C_i$ we must store the orbits of $\Gamma_i$ 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 $\Pi_i$}
    FixedPt: Integer {the point $u_i$ fixed to get $\Pi_{i+1}$}
    ItsGroup: GroupPtr {the group $\Gamma_i$}
    CellOrbits: PermPtr {the orbits of $\Gamma_i$ 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 $\Gamma_i$ on $C_i$, the first cell of $\Pi_i$, 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
same representative. The value of $\text{CellOrbits}^\wedge[v]$ is a pointer toward the representative, which in turn is marked by a negative value. We initialize $\text{CellOrbits}^\wedge[v] = -1$, for all $v \in C_i$. The entries for $v \not\in C_i$ 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 $\Gamma_i$ is found, there will be a call to $AddGen(\gamma, \Gamma_i)$. $AddGen$ must update the known orbits of $\Gamma_i$ on $C_i$. Thus it will contain a call to the following procedure, which can be placed immediately after the statement adding $\gamma$ to the list of generators.

$\textbf{UpdateOrbits}(\gamma: \text{PermPtr}, C: \text{CellPtr})$
\{update the orbits of the stabiliser on cell $C$\}

\textbf{Begin} with $C^\wedge$ do begin
\textbf{for} $k := \text{FirstPt}$ to $\text{LastPt}$ do begin
\hspace{1em} $u := V[k]$ \{ $k^{\text{th}}$ vertex of the graph\}
\hspace{1em} $v := \gamma^\wedge[u]$
\hspace{1em} $\text{uRep} := \text{ OrbitRep}(u)$ \{orbit representative for $u$\}
\hspace{1em} $\text{vRep} := \text{ OrbitRep}(v)$ \{orbit representative for $v$\}
\hspace{1em} if $\text{uRep} \neq \text{vRep}$ then $\text{Merge(uRep, vRep)}$
\textbf{end}
\textbf{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.

$\textbf{AddGen}(\gamma: \text{PermPtr}; S: \text{SearchNodePtr})$

The procedure $\text{UpdateOrbits}$ 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 $\text{CellOrbits}$ 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 $\text{CellOrbits}$ for the orbit representatives.

$\textbf{Merge}(\text{uRep}, \text{vRep}: \text{Integer})$
\{merge the orbits of $\text{uRep}$ and $\text{vRep}$, which are orbit representatives\}

\textbf{Begin}
\hspace{1em} $\text{uSize} := -\text{CellOrbits}^\wedge[\text{uRep}]$
\hspace{1em} $\text{vSize} := -\text{CellOrbits}^\wedge[\text{vRep}]$
\hspace{1em} if $\text{uSize} < \text{vSize}$ then begin

\textbf{End}
On Writing Isomorphism Programs

CellOrbits^[uRep] := vRep
  \hspace{1cm} w := vRep \quad \text{the new orbit representative}
end
else begin
  CellOrbits^[vRep] := uRep
  w := uRep \quad \text{the new orbit representative}
end
CellOrbits^[w] := -(uSize + vSize)
End \quad \{\text{Merge}\}

The function \text{OrbitRep} follows the pointers \text{CellOrbits}[v] to the orbit representative.

\text{OrbitRep}(v: \text{Integer}): \text{Integer}
  \{\text{follow the pointers to the orbit representative}\}
var \hspace{0.5cm} w: \text{Integer}
Begin
  if CellOrbits^[v] < 0 then return(v)
  w := OrbitRep(\text{CellOrbits}^/[v])
  CellOrbits^/[v] := w \quad \text{path compression}
return(w)
End \quad \{\text{OrbitRep}\}

These procedures require access to the \text{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 \text{CellOrbits} array.

Comparing Orderings.

As presented in section 3, the procedure \text{Stabilise} first refines \Pi, then selects in turn each \( u \in C \in \Pi \), creates a partition \( \Pi_u \) in which \( u \) is fixed, and calls itself recursively. During the refinement of \( \Pi_u \), 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 \text{Refine} procedure will also set the variable \( N\text{Fixed} \), the number of points currently on the array \( F \). In order to have access to this variable, we must alter the calling parameters for \text{Refine} to include the current searchnode, which contains the partition to be refined.

\text{Refine}(S: \text{SearchNodePtr})
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 initialized yet. We use a global boolean variable \( B_{\text{exists}} \) to indicate this condition. It is initially set to \( \text{false} \). In order to detect whether a refined partition is discrete, the \textit{Refine} procedure can set a global boolean variable \( \text{isDiscrete} \). The initial search node created containing the unit partition (before refinement) is saved as a global variable, called \textit{TopSearchNode}. The \textit{GroupPtr} that it contains points to \( \text{Aut}(G) \). When an automorphism is discovered by the program, it is added to the \textit{TopSearchNode}. A refined version of the procedure \textit{Stabilise} is now presented.

\begin{verbatim}
Stabilise(S: SearchNodePtr) {refined version}
    {S is the current search node, containing a partition \Pi. Refine \Pi,
     then select \mu in the first cell of \Pi in all inequivalent ways}
Begin
    m := S^\text{NFixed} {save the number of points currently fixed}
    S^\text{OnBestPath} := \text{false}
    Refine(S) {refine the partition in this search node}
    Result := CompareOrders(m + 1, S^\text{NFixed})
    if \text{isDiscrete} then begin
        if \text{B\_exists} then begin begin
            case Result of
                equal: begin {an automorphism has been found}
                    \gamma := \text{NewPerm}(n) {allocate a new PermPtr}
                    for \( k := 1 \) to \( n \) do \gamma[F[k]] := B[k]
                    AddGen(\gamma, \text{TopSearchNode})
                    AutoFound := \text{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 \( B[1..n] \)
        end
    end
End
\end{verbatim}
On Writing Isomorphism Programs

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 $S^\wedge$ do begin
   $\Pi$ := Partition
   if NextSearchNode=nil then create and initialize
      NextSearchNode
   $S_u$ := NextSearchNode
end
$C$ := first cell of $\Pi$
$u$ := $C^\wedge$.FirstPt
repeat  {until all inequivalent choices $u$ have been made}
   $S_u^\wedge$.FixedPt := $u$
   make a copy $\Pi_u$ of $\Pi$ in which $C$ is split into \{u\} and
      $C$ \{-u\}
   $S_u^\wedge$.Partition := $\Pi_u$
   Stabilise($S_u$)
   if AutoFound then ...
      dispose($\Pi_u$)
      mark CellOrbits$^\wedge[u]$ to indicate that $u$ has been fixed
      change the base of Aut($G$) if necessary
      select next inequivalent $u \in C$ to stabilise
   until no $u$ was found 1:
      {reset degrees to 0 before returning}
      for $k$ := $m + 1$ to $S^\wedge$.NFixed do Degree$^\wedge[F[k]]$ := 0
      $S^\wedge$.FixedPt := 0  {mark this search node inactive}
End  {Stabilise}
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 $u$  
$\text{isDiscrete}$: Boolean, whether refinement produced a discrete partition  
$B\text{.exists}$: Boolean, whether the array $B$ has a value  
$\text{AutoFound}$: Boolean, whether an automorphism was found  
$\text{TopSearchNode}$: SearchNodePtr, the initial search node  
$\text{LastBaseChange}$: SearchNodePtr, see below  
$\text{BasisOK}$: 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.

$\text{Stabilise}(\Pi_0)$ is called, $\Pi_0$ is refined  
$\Pi_0 = \{1, 2, 3, 4, 5, 6\}$ is equitable  
fix $1 \Rightarrow \quad \Pi_1 = \{1 \mid 2, 3, 4, 5, 6\}$  
$\text{Stabilise}(\Pi_1)$ is called, $\Pi_1$ is refined  
$\Pi_1 = \{1 \mid 6 \mid 2, 3, 4, 5\}$ is equitable, $F = (1, 6)$  
fix $2 \Rightarrow \quad \Pi_2 = \{1 \mid 6 \mid 2 \mid 3, 4, 5\}$  
$\text{Stabilise}(\Pi_2)$ is called, $\Pi_2$ is refined  
$\Pi_2 = \{1 \mid 6 \mid 2 \mid 4 \mid 3, 5\}$ is equitable, $F = (1, 6, 2, 4)$  
fix $3 \Rightarrow \quad \Pi_3 = \{1 \mid 6 \mid 2 \mid 4 \mid 3 \mid 5\}$  
$\text{Stabilise}(\Pi_3)$ is called, $\Pi_3$ is refined
On Writing Isomorphism Programs

\[ \Pi_3 \text{ is discrete, } B = (1, 6, 2, 4, 3, 5) \text{ is assigned} \]

Backtrack to \( \Pi_2 \)

fix 5 \( \Rightarrow \) \( \Pi_3 = \{1 \mid 6 \mid 2 \mid 4 \mid 5 \mid 3\} \)

\text{Stabilise}(\Pi_3) \text{ is called, } \Pi_3 \text{ is refined}

\( \Pi_3 \text{ is discrete, } F = (1, 6, 2, 4, 5, 3) \)

An automorphism \( \gamma = (3, 5) \) is discovered.

\text{AddGen} \text{ is called.}

The orbits of each \( \Gamma_i \) on the first cell of \( \Pi_i \) 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 \( \Pi_2 \)

In \( \Pi_2 \) both 3 and 5 have already been fixed

\( \Rightarrow \) backtrack to \( \Pi_1 \)

In \( \Pi_1 \), only 2 has been fixed so far

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

\text{Stabilise}(\Pi_2) \text{ is called, } \Pi_2 \text{ is refined}

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

fix 2 \( \Rightarrow \) \( \Pi_3 = \{1 \mid 6 \mid 3 \mid 5 \mid 2 \mid 4\} \)

\text{Stabilise}(\Pi_3) \text{ is called, } \Pi_3 \text{ is refined}

\( \Pi_3 \text{ is discrete, } F = (1, 6, 3, 5, 2, 4) \)

An automorphism \( \gamma = (2, 3)(4, 5) \) is discovered.

\text{AddGen} \text{ is called.}

The generators of \( \text{Aut}(G) \) are now \( (3, 5) \) and \( (2, 3)(4, 5) \)

The orbits of each \( \Gamma_i \) on the first cell of \( \Pi_i \) 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 \( \Pi_2 \)

In \( \Pi_2 \) points 2 and 4 are in the same orbit

\( \Rightarrow \) backtrack to \( \Pi_1 \)

In \( \Pi_1 \) points 2, 3, 4, 5 are in the same orbit

\( \Rightarrow \) backtrack to \( \Pi_0 \)

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

\text{Stabilise}(\Pi_1) \text{ is called, } \Pi_1 \text{ is refined}

\( \Pi_1 = \{2 \mid 4 \mid 1, 3, 5, 6\} \text{ is equitable, } F = (2, 4) \)
fix 1 \Rightarrow \quad \Pi_2 = \{2 \mid 4 \mid 1 \mid 3, 5, 6\}

Stabilise(\Pi_2) is called, \Pi_2 is refined

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

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

Stabilise(\Pi_3) is called, \Pi_3 is refined

\Pi_3 is discrete, \quad F = (2, 4, 1, 6, 3, 5)

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

AddGen is called.

The generators of Aut(G) are now

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

The orbits of each \Gamma_i on the first cell of \Pi_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 \Pi_2

In \Pi_2 points 3 and 5 are in the same orbit

\Rightarrow \text{backtrack to } \Pi_1

In \Pi_1 points 1, 3, 5, 6 are in the same orbit \Rightarrow \text{backtrack to } \Pi_0

In \Pi_0 points 1, 2, 3, 4, 5, 6 are in the same orbit \Rightarrow \text{done.}

The program visited 4 leaf nodes in the search tree, as shown in Fig. 7. Three generators for Aut(G) were found. It is easy to see that these generators give all of Aut(G), for the following reason. If the orbits of \Gamma_i on C_4 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 |Aut(G)|. Since every one of these would be visited by the search, every automorphism would be found. By computing the orbits of \Gamma_i on C_4, 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 \Gamma_i in \Gamma_{i-1}. Consequently the generators found generate all of Aut(G). The group \Gamma_0 is initially the identity group, and grows until it equals Aut(G). \Gamma_0 is always a subgroup of Aut(G). So each time a new generator is discovered, \Gamma_0 must at least double in size. Therefore the number of generators found is at most \log_2|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
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.
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 "$\text{CompareOrders}(m + 1, 5'.\text{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 $B_{\text{exists}}$ 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 $\gamma$ that was discovered maps each $u_i$ to $v_i$. Therefore
On Writing Isomorphism Programs

Once $\gamma$ has been added to $\text{Aut}(G)$, 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 $\text{OnBestPath}$. This is initialized to $\text{false}$ each time $\text{Stabilise}$ is entered. Whenever $B$ is assigned, the program executes a loop which starts at $\text{TopSearchNode}$ and sets the value of $\text{OnBestPath}$ to $\text{true}$ for every search node. When the algorithm later descends into other branches of the search tree, some of these will be changed to $\text{false}$. 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 $\text{AutoFound}$ is set to $\text{true}$. So long as $\text{AutoFound}$ remains $\text{true}$ and $\text{OnBestPath}$ is $\text{false}$, 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 $\text{Stabilise}$.

```
if $\text{AutoFound}$ then begin
    if not $S^\ast . \text{OnBestPath}$ then goto 1
    $\text{AutoFound} := \text{false}$
end
```

When $\text{Stabilise}$ is descending the search tree the values $\text{Degree}[v]$ are not re-initialized after each refinement. This is because the degree is a cell-invariant of the partition. Before returning to the point where $\text{Stabilise}$ was called from, it must re-initialize 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-initialization 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.
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 $\Pi_i$. A vertex $u_i \in C_i$ is to be selected for stabilisation. We need to know the orbits of $\Gamma_i$ 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 $\text{Aut}(G)$.

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 $\gamma$. This must be changed so that the basis of the group always agrees with that of the partition.

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

Sometimes the basis of the group will be longer than the basis of the partition. In order to accommodate 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 $\text{Aut}(G)$ 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 $\text{Aut}(G)$, 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 ascends the search tree up to a partition $\Pi_i$, the bases will still agree up to depth $i$. If the program now selects a new $u_i$ to stabilise and descends to $\Pi_{i+1}$ 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
On Writing Isomorphism Programs

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 ...

\[
\begin{align*}
\text{if } S^\wedge.\text{Depth} > \text{BasisOK then } & \text{ChangeBase}(S^\wedge.\text{Depth}) \\
\text{BasisOK} := S^\wedge.\text{Depth} & \\
\text{LastBaseChange} := S &
\end{align*}
\]

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-initializes 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.

\textbf{ChangeBase}(d: \text{Integer})

\{change the basis of the automorphism group, called from depth d\}

\text{var} \quad \Gamma: \text{GroupPtr} \\
\quad S: \text{SearchNodePtr} \\
\text{Begin} \\
\quad S := \text{LastBaseChange} \\
\quad \Gamma := S^\wedge.\text{ItsGroup} \\
\quad \text{if } \Gamma = \text{nil then return } \{\text{no group at this depth of recursion}\} \\
\quad \text{if } \Gamma^\wedge.\Gamma_u = \text{nil then return } \{\Gamma = \text{identity group}\} \\
\quad \{\text{first reset the cell orbits from depth BasisOK}\} \\
\quad S := S^\wedge.\text{NextSearchNode} \\
\quad \text{while } S \neq \text{nil do begin} \\
\quad \{\text{search nodes below depth } d \text{ do not correspond to partitions/}\} \\
\quad \text{if } S^\wedge.\text{Depth} \leq d \text{ then begin} \\
\quad \quad C := \text{first cell of } S^\wedge.\text{Partition} \\
\quad \quad \text{for } i := C^\wedge.\text{FirstPt} \text{ to } C^\wedge.\text{LastPt} \text{ do} \\
\quad \quad \quad S^\wedge.\text{CellOrbits}^\wedge[V[i]] := -1 \\
\quad \text{end} \\
\text{end}
Marking the Cell Orbits.

When a point \( u_i \in C_i \) has been stabilised in the search node containing \( \Pi_i \), the orbit of \( \Gamma_i \) 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 \( \text{CellOrbits} \) array. Currently the value stored in \( \text{CellOrbits}[v] \) has the following property.

\[
\begin{align*}
\text{CellOrbits}[v] &> 0, \text{ if } v \text{ is not the representative of } \text{Orb}(v) \\
\text{CellOrbits}[v] &= -|\text{Orb}(v)|, \text{ if } v \text{ is the representative of } \text{Orb}(v)
\end{align*}
\]

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

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

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-initialized every time the basis is changed.
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


On Writing Isomorphism Programs


