Groups & Graphs, a MacIntosh Application for Graph Theory

William Kocay [1]
Department of Computer Science
University of Manitoba
Winnipeg, CANADA R3T 2N2

Abstract

Groups & Graphs is a research tool for computing with graphs and their automorphism groups. This note describes the various kinds of information that it can provide.

Groups & Graphs, a MacIntosh application, is a research tool for manipulating graphs on a computer screen, and for computing with them and their automorphism groups. Groups & Graphs can be used by researchers as a graph- and group-calculator, in a manner similar to the Cayley [4,5] group-theoretical programming package. It does not have the enormous scope of Cayley, but it is portable, and it is quite diversified. The purpose of this article is to describe the structure of the program and some of the algorithms that it uses.

The package consists of three main parts : graph editing and manipulation, graph algorithms, and group algorithms. The automorphism group computation is the bridge between the latter two parts. Once the automorphism group of a graph has been calculated, computation on the group will often reveal properties of the graph. Graph algorithms which are currently implemented include : maximum matching, finding a long path, computing the automorphism group, determining whether two graphs are isomorphic, computing the line-graph, choosing a subgraph, finding the cosets of the subgraph, and symmetrizing under the action of a given group. Group algorithms which are currently available include : finding the group generated by a set of permutations, listing the elements of the group, finding the commutator subgroup and point-stabiliser subgroups, finding a block system for the group, computing the subgroup fixing any given block, finding the homomorphic image in which the blocks are permuted as "points", computing and listing a representative of each coset of a given subgroup, and listing the elements of a particular coset. The importance of the package is that it provides easy access to the kinds of computations one normally wants

[1] e-mail: BKOCAY@UOFMCC.bitnet. This work was supported by an operating grant from the Natural Sciences and Engineering Research Council of Canada.

to do for graphs and their groups; it acts like a hand calculator designed especially for graphs and groups. Results of computations are usually displayed pictorially (a picture is worth 1,000 words [13]).

Let X be a graph with vertex set V(X) and edge set E(X). The graph-theoretic notation used will mostly follow that of Bondy and Murty [3]. We shall often just write X for V(X), especially when a group is acting on the graph, and shall set $n = |V(X)|$. We only consider *simple* graphs, that is, no multiple edges or loops are allowed; thus $E(X) \subseteq \binom{X}{2}$, the set of all 2-subsets of X. Let G be a permutation group acting on a set X of *points*. X will often be the vertex-set of a graph, so that vertex and point will be used as synonyms. Given any $x \in X$ and any $\theta \in G$, x^θ denotes the point of x under the mapping θ. When X also represents a graph, G is a subgroup of the *automorphism* group of X if its induced action on 2-subsets fixes E(X), that is, $E(X)^G = E(X)$. Aut(X) denotes the group consisting of all automorphisms of X. It is a subset of the *symmetric* group Sym(X) acting on X.

Each graph or group displayed by the program is associated with a window on the screen. A graph window is illustrated in Fig. 1.

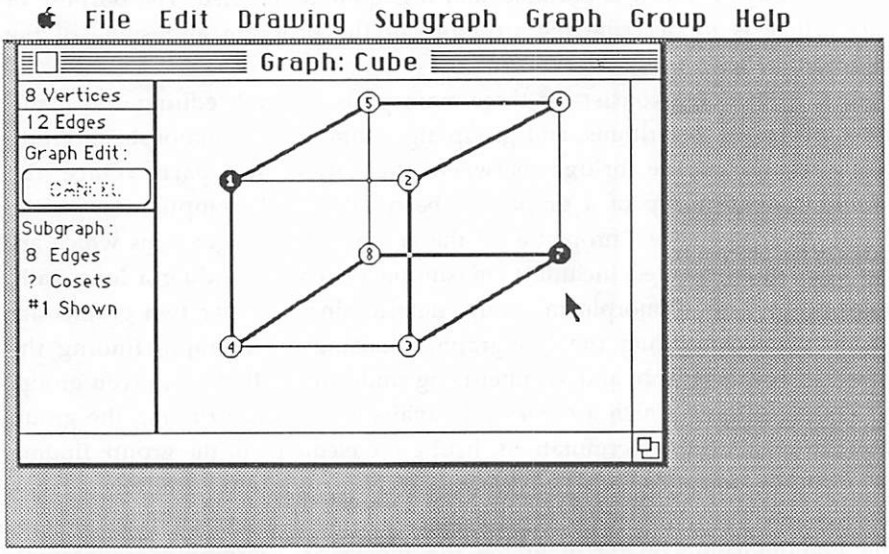

Fig. 1 : A typical graph window.

The window shows the name of the graph and the number of vertices and edges, as well as some other information. It is always possible to highlight a particular subgraph, as illustrated. The edges of the chosen subgraph are drawn with thicker lines. The currently chosen subgraph will always be

denoted by Z. The graph window also indicates the number of edges in Z and some other subgraph information. Vertices of the graph can also be selected, in which case they are drawn in black. The selected set of vertices is denoted by S. The graph can be edited in several ways:

1) A *walk* in V(X) is any sequence of vertices $v_0v_1v_2...v_k$. A walk can be chosen by clicking from node to node. For pairs $v_i v_{i+1} \notin E(X)$, that is, those not defining an edge in X, an edge is inserted, and for pairs $v_i v_{i+1} \in E(X)$, the edge is deleted, as the walk is traced.

2) A *clique* (complete subgraph) K_S can be induced on the selected vertices S.

3) A *bi-clique* (complete bipartite subgraph) $K_{S,X-S}$ can be induced between S and X-S.

4) An *anti-clique* (independent or stable set of vertices) can be induced on S.

5) A new vertex can be inserted, joined to all selected vertices.

6) X[S], the subgraph *induced* by S, can be replaced by its complement, K_S-X[S].

7) The graph can be *switched* on the set S, that is, the set [S,X-S] of edges consisting of all edges with one end in S and one in X-S, is removed, and replaced by the complementary set $K_{S,X-S}$-[S,X-S].

8) The edges of the subgraph Z can be deleted from X.

9) The edges of Z can all be subdivided by inserting a new vertex of degree 2 on each edge of Z.

10) The selected vertices S can be *identified* as one new vertex s ; s then replaces S. Any multiple edges and loops created are deleted.

The subgraph Z can also be edited :

11) One may choose a walk $v_0v_1v_2...v_k$ in X so that, for each $v_i v_{i+1} \in E(X)$, $v_i v_{i+1}$ is added to E(Z) if $v_i v_{i+1} \notin E(Z)$, and is removed from E(Z) if $v_i v_{i+1} \in E(Z)$.

12) Z can be chosen as the induced subgraph X[S].

13) The complement of Z relative to X can be chosen as Z, that is, Z := X-Z.

14) Z can be chosen as the *edge-cut* [S,X-S]. In case S={v}, this is useful for finding the *neighbourhood* v* of v, that is, those vertices adjacent to v.

Graph Algorithms.

Most of the algorithms currently available produce a subgraph Z as the result of their application to X. This makes the result of applying an algorithm immediately visible. Balanced incomplete block designs [16] can be

conveniently dealt with by representing them as bipartite graphs, with points versus blocks.

Maximum Matching.

A *matching* $M \subseteq E(X)$ in X is a set of edges such that each $u \in V(X)$ is incident with at most one edge of M. An $O(n^3)$ version of the Edmonds matching algorithm [6] is used to find a maximum matching in X. The implementation programmed is similar to that desribed in [17]. If X happens to be bipartite, this is equivalent to the Hungarian algorithm [3].

Hamilton Paths and Cycles.

A Hamilton path in X is a path of length n-1, and a Hamilton cycle is a cycle of length n. The problem of determining whether X has either a Hamilton path or a Hamilton cycle is NP-complete, but there is a very simple $O(n^2)$ algorithm [7] which will often find a Hamilton path or cycle if one exists. Beginning at any vertex, a path P as long as possible is constructed, without backtracking, until it can no longer be extended. We then know that the endpoints of this path are joined only to other vertices of P. If the special configuration involving the endpoints shown in Fig. 2 exists in the graph, then the path P can be converted into a cycle C. Since X is connected, some $u \in C$ will be joined to $v \notin C$; this produces a new path P' that is longer than P. We now work with the path P', extending it in both directions and repeating the above steps as often as necessary, until no further improvement is possible.

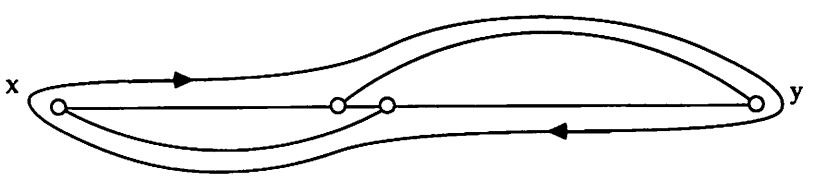

Fig. 2 : Converting a path to a cycle.

Line Graph and Complement.

The *line graph* L(G) has a vertex for every $uv \in E(G)$ with two edges uv and xy adjacent if they share a common endpoint. A window for the line graph can be constructed, as shown in Fig. 3 for the graph of the cube. Similarly, G can be replaced by its complement.

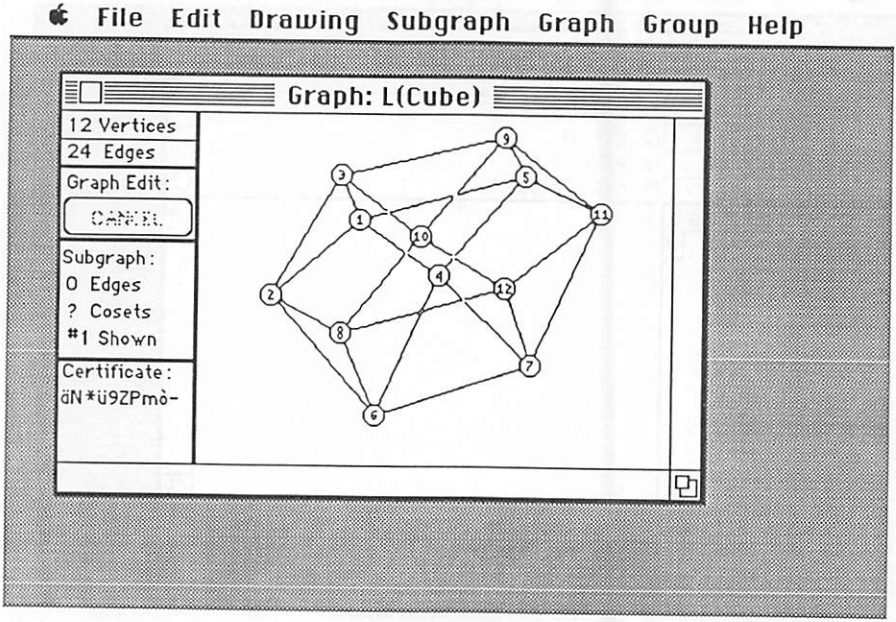

Fig. 3 : Line graph of the cube.

Automorphism Group.

The algorithm used to compute Aut(X) is a version of that described in [10] ; it has been rewritten for the MacIntosh to improve the efficiency. It uses partition refinement (naively called *naive* refinement by some authors), which is still the best and fastest method known for general graph isomorphism [9,14]. It produces generators for Aut(X) and a *certificate*, Cert(X), for X. A certificate is an encoding of X, ususally a character string, such that X is isomorphic to Y (written $X \cong Y$) if and only if Cert(X) = Cert(Y). Cert(X) also appears in the window for X, as illustrated in FIg. 3. Once Aut(X) has been computed, a group window is created to display it, as described later.

Group Algorithms.

A group window contains a group G. The automorphism group of the cube is displayed in Fig. 4.

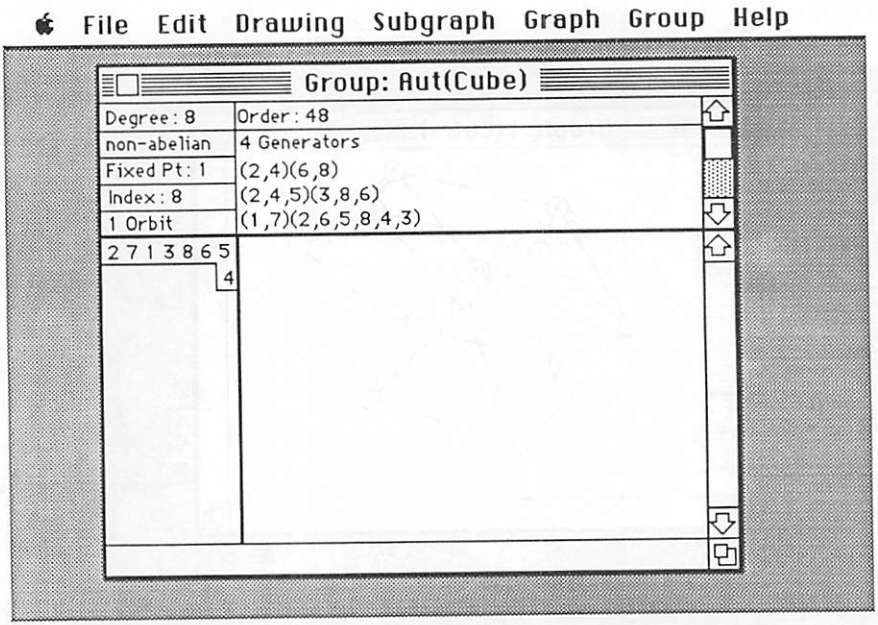

Fig. 4 : A group window.

The window displays a list of generators for G, the orbits of G, the order of G, and whether or not G is abelian. We can list the elements of any group of order $\leq 2^{31}-1$ in the group window. This is done by representing a group G in terms of the cosets of a stabiliser subgroup:

```
PermPtr = ^Perm;
Perm = array of Integer;
GroupPtr = ^Group
Group = record
    Generators: linked list of PermPtr;  { generators for the group }
    Subgroup: GroupPtr;  { a stabiliser subgroup }
    Rep: Integer;  { the point whose stabiliser is the Subgroup }
    Orbit: array of Integer; { the orbit of Rep }
    OrbitSize: Integer; { length of  Orbit }
    Transversal: array of  PermPtr  { coset reps for Subgroup cosets }
  end;
```

With this data structure, we can recursively list the elements of each coset of the stabiliser subgroup. It is also very easy to determine whether a given permutation is an element of a group G. We present the Pascal-like pseudo-

code for it.

```
Function GroupElement(G: GroupPtr;  P: PermPtr): Boolean;
var Q: PermPtr;
Begin
   GroupElement := true;
   if P=IdentityPerm then return;  { the identity is in every group }
   GroupElement := false;
   with G^ do begin
      If Generators=nil then return;
      k := P^[Rep];
      if k∉ Orbit then return;
      Q := P*Inverse(Transversal[k]); { permutation multiplication }
      GroupElement := GroupElement(Subgroup, Q)
   end
End;
```

Block Systems.

A *block system* [8] for a transitive group G acting on X is a partition of X into disjoint sets B_1, B_2, ..., B_k such that every element of G induces a permutation of B_1, B_2, ..., B_k. If a group G has several orbits, they will be displayed in the window. An orbit can be *selected*, by "clicking" on it. Once an orbit O has been selected, we can find a block system for G acting on this orbit (the *transitive constituent*, G[O], of G acting on O). It will be displayed graphically in the window, as shown in Fig. 5 for the line graph of the cube.

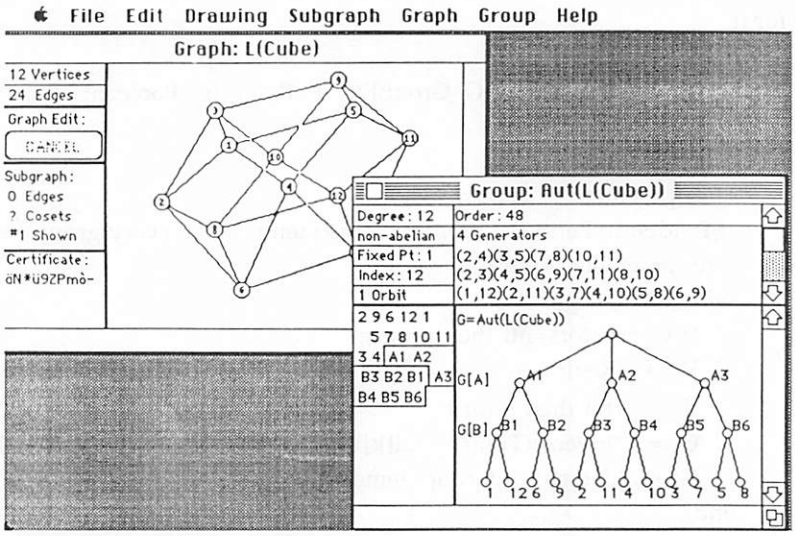

Fig. 5 : Window showing a block system for G.

G will induce a group G_B acting on the blocks B_1, B_2, ..., B_k. It may also have a block system, etc., so that the various hierarchical block systems form a tree. The tree of block systems will be displayed in the window. If G happens to be Aut(X) for a graph X, then the block systems of G will often provide insight into the structure of X, as the vertices of X can only be permuted in blocks. The algorithm used to find the tree of block systems is an $O(n^2 \log n)$ version of that presented by Atkinson [1].

Once a tree of block systems has been found, each set of blocks will appear in the window as orbits of G, and these orbits can also be selected. The operation to construct the transitive constituent for an orbit O will now compute G_B, the induced group acting on the blocks. Alternatively, a particular block B_i can be chosen and the subgroup fixing B_i can be computed.

Subgroups.

As well as stabiliser subgroups and block-stabilisers, Groups & Graphs computes the commutator subgroup of G. It is generated by all

commutators, $[a,b]=a^{-1}b^{-1}ab$, where $a,b \in G$.

All subgroups of G computed are saved in a list. If K is a particular subgroup, then G acts on the right cosets of K by the rule $g \in G$: $Ka \rightarrow Kag$. The resulting permutations of the cosets form the factor group G mod K, [2, Ch. 8], of degree [G:K]. Once a subgroup K has been selected, G mod K can be computed, although this computation is currently limited to subgroups of index $[G:K] \leq 127$, because a permutation is stored as an array of signed bytes. G is first decomposed into cosets: $G=Ka_1 + Ka_2 + ... + Ka_r$. The coset representatives a_i can also be listed in the group window. The algorithm used to compute the coset representatives is a version of Sims's algorithm [15], which we describe briefly. It is based on the idea of ordering the elements of G, and choosing the smallest element of a coset Ka as a *canonical* representative of Ka. Let G act on $X=\{1,2,...,n\}$. Let G_x denote the stabiliser subgroup of G fixing point $x \in X$, with coset decomposition $G = G_xb_1 + G_xb_2 + ... + G_xb_k$. Each coset G_xb_i corresponds to the point x^{b_i} in the orbit of x. If we scan the orbit of X, which is stored in the data structure representing a group, we can choose the smallest such point x^{b_i} in the orbit. This defines an ordering of the stabiliser cosets G_xb_i. We now proceed recursively, and choose the smallest coset of a 2-point stabiliser $G_{x,y}$, contained in G_xb_i, etc., until we come to cosets containing only one permutation. This defines the smallest element of the group G. It can also be used to find the smallest element of any coset Ka of a subgroup K, just by using the stabiliser subgroups and orbits stored in the data structure representing K. The Pascal-like pseudo-code for this follows.

```
CosetRep(K: GroupPtr; g: PermPtr): PermPtr;
{ K is a subgroup of a group G. Computes g', the minimum element of
  the coset Kg }
var a, MinPerm: PermPtr;
    x, MinPt: Integer;
Begin
   CosetRep := g;
   with K^ do begin
      if Generators=nil then return;   {K is the identity subgroup }
      { select the min element in the orbit }
      MinPt := n;
      for each x∈Orbit do begin
         a := Transversal[x]; { the coset rep for point x }
         { check coset Subgroup.ag }
         if g^[a^[Rep]] < MinPt then begin
```

```
                MinPt:= g^[a^[Rep]];
                MinPerm := a
              end
          end;
          { the g' we want lies in the coset  Subgroup. MinPerm.g  }
          a := MinPerm * g ;  { permutation multiplication }
          CosetRep :=  CosetRep(Subgroup, a))
      end
  End;
```

Groups & Graphs will also list the elements of any coset Ga or Ka of a group G or subgroup K, where a is any element of the symmetric group Sym(X).

Groups and Graphs.

Once G = Aut(X) has been computed, certain computations in which the group G and the graph X interact are available.

Listing Isomorphisms.

Computing Aut(X) also causes a certificate Cert(X) to be computed. If two graphs, X and Y, are on the screen and both automorphism groups have been computed, then the set of all isomorphisms from X to Y is a coset $Aut(X)\theta$ of G, where θ is any given isomorphism from X to Y. Groups & Graphs will compute θ and then list the entire coset in the window for G. Now it may happen that X was constructed by deleting one or more vertices from a larger graph. When vertices are deleted, the remaining vertices are not renumbered, but preserve their original numbering. So the vertices of X may not be numbered 1, 2, ..., n, but may be numbered $V(X)=\{u_1, u_2, ..., u_n\}$; also, the vertices of Y may be numbered $V(Y)=\{v_1, v_2, ..., v_n\}$. So the isomorphism θ is not a permutation; thus the mappings $Aut(X)\theta$ are not permutations either. We can treat them as *partial permutations*[11,12]. Each $v_i \notin V(X)$ has no image under θ. Each $u_i \notin V(Y)$ is not an image. This means that, unlike permutations, in which points fall into disjoint cycles, the points of partial permutations fall into disjoint *paths and cycles*; for example, $\theta=<1,3,6,4>(2,5,7)$ means that (2,5,7) forms a cycle, but that <1,3,6,4> forms a path for which 4 has no image and 1 is not an image. So every $u_i \notin V(Y)$ begins a path and every $v_i \notin V(X)$ ends a path. The mappings of $Aut(X)\theta$ are printed in this disjoint path and cycle notation.

Subgraph Cosets.

If the graph X has a subgraph Z specified, then the automorphism computation calculates K = Aut'(X), the subgroup of G = Aut(X) that maps Z to Z. Each coset Ka of K in G now consists of all mappings taking the subgraph Z to an isomorphic subgraph Z^a. This is called a *subgraph coset*. Groups & Graphs will also compute a list of all the subgraph cosets of Z and will display them, one after the other.

Symmetrization.

If a graph X and a group G are both on the screen so that every point of the group is also a point of the graph, then in certain cases we can allow G to act on X, so as to *symmetrize* it. Groups & Graphs will compute the orbit, $E(X)^G$, of the edge-set of X under G and will add all these edges to X if they are not already present. Thus Aut(X) will now include G as a subgroup. Suppose that there are one or more points v of X which are not also points of G. Let v^* denote the *neighbourhood* of v, that is, all points adjacent to v, and suppose that every point of v^* is also a point of G. The orbit $(v^*)^G$ of the neighbourhood will be computed, and a new point u will be added to X for every new set $U \in (v^*)^G$ in such a way that $u^*=U$. This can be useful for extending graphs in a symmetric way or for producing block designs using difference sets. For example if we begin with a graph X with 7 vertices, {1,2,...,7}, and no edges, and then add a point adjacent to {1,2,4}, allowing the group G generated by the cycle (1,2,3,4,5,6,7) to act on X, we produce the *Heawood* graph [3], which is the point-block incidence graph of the Fano plane. Once the right regular representation of a group G has been computed, symmetrizing can also be used to construct Cayley graphs for G.

There are a great many other features which would be convenient in a program for calculating with graphs and their groups. Some of these are currently being added to the program. Addition of several others is in the planning stage. Clearly there is a practical limit on the size to which the application can reasonably grow. This note is published for the convenience of those who may be interested in the program. The actual application is available upon request from the author.

References

1. M. Atkinson, An algorithm for finding the blocks of a permutation group, Maths. of Computation 29, 1975, pp. 911-913.
2. B. Bollobás, *Graph Theory, an Introductory Course*, Springer-Verlag, New York, 1979.
3. J.A. Bondy and U.S.R. Murty, *Graph Theory with Applications*, American Elsevier Publishing Co., New York, 1976.
4. Greg Butler and John Cannon, The Cayley system for discrete algebraic and combinatorial structures, preprint, Department of Computer Science, University of Sydney, 1988.
5. J.J. Cannon, An introduction to the group theory language Cayley, in *Computational Group Theory*, edited by M.D. Atkinson, Academic Press, London, 1984, pp. 145-183.
6. J. Edmonds, Paths, trees, and flowers, Canadian J. Maths. 17, 1965, pp. 449-467.
7. Shimon Even, *Graph Algorithms*, Computer Science Press, Potomac, Maryland, 1979,
8. Marshall Hall, Jr., *The Theory of Groups*, Chelsea Publishing Co., New York, 1976.
9. Andrew J. Kirk, Efficiency Considerations in the Canonical Labelling of Graphs, B.Sc. Thesis, ANU, Canberra, 1985.
10. William Kocay, Abstract data types and graph isomorphism, J. Combinatorics, Information, and System Sciences, 1984, pp. 247-259.
11. W.L. Kocay, Graphs, groups, and pseudo-similar vertices, J. Austral. Math. Soc. (A) 37, 1984, pp. 181-189.
12. W.L. Kocay, Hypomorphisms, orbits, and reconstruction, J. Comb. Th. (B), 1987, to appear.
13. K'ung Chiu (Confucius), *The Analects*, circa 500 B.C.
14. Brendan McKay, Practical graph isomorphism, Congressus Numerantium 30, 1981, pp. 45-87.
15. Charles C. Sims, Computation with permutation groups, ACM Second Symposium on Symbolic and Algebraic Manipulation, Ed. S.R. Petrick, 1971
16. Anne Penfold Street and Deborah J. Street, *Combinatorics of Experimental Design*, Oxford University Press, New York, 1987.
17. Robert Endre Tarjan, *Data Structures and Network Algorithms*, SIAM, Philadelphia, 1983.