Graph Algorithms in Chemical Computation

ROBERT ENDRE TARJAN*

Computer Science Dept., Stanford University, Stanford, CA 94305

1. Introduction.

The use of computers in science is widespread. Without powerful number-crunching facilities at his** disposal, the modern scientist would be greatly handicapped, unable to perform the thousands or millions of calculations required to analyze his data or explore the implications of his favorite theory. He (or his assistant) thus requires at least some familiarity with computers, the programming of computers, and the methods which might be used by computers to solve his problems. An entire branch of mathematics, numerical analysis, exists to analyze the behavior of numerical algorithms.

However, the typical scientist's appreciation of the computer may be too narrow. Computers are much more than fast adders and multipliers; they are symbol manipulators of a very general kind. A scientist who writes programs in FORTRAN or some similar, scientifically oriented computer language, may be unaware of the potential use of computers to solve computational, but not necessarily numeric, problems which might arise in his research.

This paper discusses the use of computers to solve non-numeric problems in chemistry. I shall focus on a particular problem, that of identifying chemical structure, and examine computer methods for solving it. The discussion will include

---

* This research was partially supported by the National Science Foundation, grant MCS75-22870, and by the Office of Naval Research, contract N00014-76-C-0688.

** For the purpose of smooth reading, I have used the masculine gender throughout this paper.

Christoffersen; Algorithms for Chemical Computations
ACS Symposium Series; American Chemical Society: Washington, DC, 1977.

elements of graph theory, list processing, analysis of algorithms, and computational complexity. I write as a computer scientist, not as a chemist; I shall neglect details of chemistry in order to focus on issues of algorithmic applicability, simplicity, and speed. It is my hope that some readers of this paper will become interested in applying to their own problems in chemistry the methods developed in recent years by computer scientists and mathematicians.

The paper is divided into several sections. Section 2 discusses representation of chemical molecules as graphs. Section 3 covers complexity measures for computer algorithms. Section 4 surveys what is known about the structure identification problem in general. Section 5 solves the problem for molecules without rings. Section 6 gives a method for analyzing a molecule by systematically breaking it into smaller parts. Section 7 discusses the case of "planar" molecules. Section 8 outlines a complete method for structure identification, and mentions some further applications of the ideas contained herein to chemistry.

2. Molecules and Their Representation.

Consider a hypothetical chemical information system which performs the following tasks. If a chemist asks the system about a certain molecule, the system will respond with the information it has concerning that molecule. If the chemist asks for a listing of all molecules which satisfy certain properties (such as containing certain radicals), the system will respond with all such molecules known to it. If the chemist asks for a listing of possible molecules (known or not), which satisfy certain properties, the system will provide a list.

Such an information system must be able to identify molecules on the basis of their structure. Given a molecule, the system must derive a unique code for the molecule, so that the code can be looked up in a table and the properties of the molecule located. It is this coding or cataloging problem which I want to consider here. A number of codes for molecules have been proposed and used; e.g. see (1,2,3,4). The existence of many different codes with no single standard suggests the importance and the difficulty of the problem. I shall attempt to explain why the problem is difficult, and to suggest some computer approaches to it.

To deal with the problem in a rigorous fashion, we couch it within the branch of mathematics called graph theory. A graph $G = (V,E)$ is a finite collection V of vertices and a finite collection E of edges. Each edge (v,w) consists of an unordered pair of distinct vertices. Each edge and each vertex may in addition have a label specifying certain information

about it. We represent a chemical molecule as a graph by constructing one vertex for each atom and one edge for each chemical bond; a ball-and-stick model of a molecule is really a graph representation of it. We label each vertex with the type of atom it represents. See Figure 1 for an example.

Two vertices v and w of a graph are said to be adjacent if (v,w) is an edge of the graph. If (v,w) is an edge, and v is a vertex contained in it, the edge and vertex are said to be incident. Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are said to be isomorphic if their vertices can be identified in a one-to-one fashion so that, if v_1 and w_1 are vertices in G_1 and v_2 and w_2 are the corresponding vertices in G_2, then (v_1, w_1) is an edge of G_1 if and only if (v_2, w_2) is an edge of G_2. Furthermore the pairs v_1, v_2; w_1, w_2; and (v_1, w_1), (v_2, w_2) must have the same labels if the graphs are labelled.

The problem we shall consider is this: given two graphs, determine if they are isomorphic. Or: given a graph, construct a code for it such that two graphs have the same code if and only if they are isomorphic. Notice that this mathematical abstraction of chemical structure identification neglects some details of chemistry. For instance, we allow bonds between only two molecules, thereby precluding the representation of resonance structures, and we ignore issues of stereochemistry (if two bonds of a carbon atom are fixed, our model allows free interchanging of the other two, whereas in the real world such interchanging may produce stereoisomers; see Figure 2). However, these are differences of detail only, which can easily be incorporated into the model; we neglect them only for simplicity. Note also that our model does not allow loops (edges of the form (v,v)), but it does allow multiple edges (which may be used to represent multiple bonds, or for other purposes).

A generalization of the isomorphism problem is the subgraph isomorphism problem. Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, we say G_1 is a subgraph of G_2 if V_1 is a subset of V_2 and E_1 is a subset of E_2. The subgraph isomorphism problem is that of determining if a given graph G_1 is isomorphic to a subgraph of another given graph G_2. This is one of the problems our hypothetical information system must solve to provide a list of molecules containing certain radicals. We shall deal with this problem briefly; it seems to be much harder than the isomorphism problem.

If a computer is to efficiently encode molecules it must first have a way to represent a molecule, or a graph. We consider

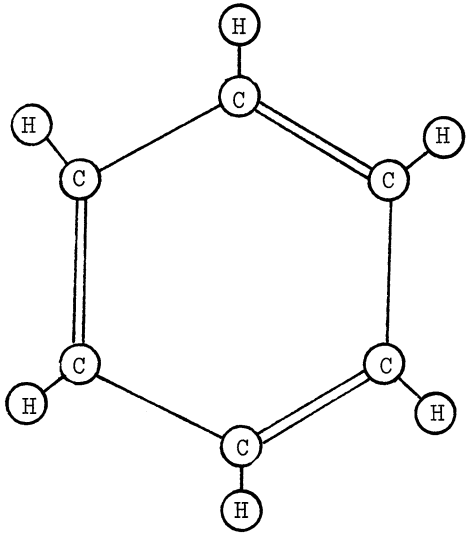


Figure 1. Graphic representation of benzene

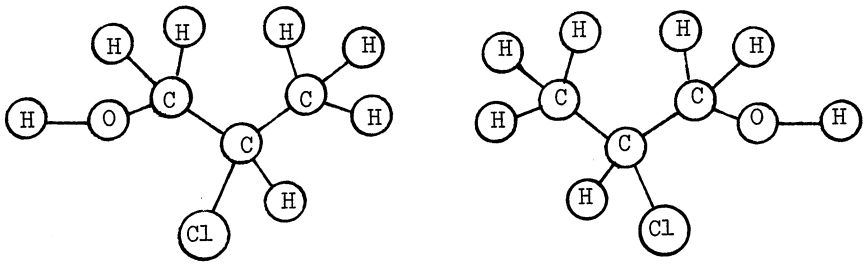


Figure 2. Stereoisomers

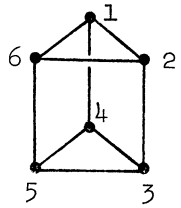
two standard ways to represent graphs in a computer. The first is by an adjacency matrix. If $G = (V,E)$ is a graph with n vertices numbered from 1 to n, an adjacency matrix for G is the n by n matrix $M = (m_{ij})$ with elements 0 and 1, such that $m_{ij} = 1$ if (v_i,v_j) is an edge of G and $m_{ij} = 0$ otherwise. See Figure 3(a), (b). Note that M is symmetric and that its main diagonal is zero. The matrix M is not a code for G since it is not unique; it depends upon the vertex numbering.

An adjacency matrix representation of a graph has several nice properties. Many natural graph operations correspond to standard matrix operations (see (5) for some examples). The bits of M can be packed in groups into computer words, so that storage of M requires only n^2/w words, if w is the word length of the machine (or only $n^2/2w$ words, if advantage is taken of the symmetry of M). If M is packed into words in this way, the bits can be processed w at a time, at least in certain kinds of computations.

However, the matrix representation has some serious disadvantages. An important property of graphs representing chemical molecules is that they are sparse; most of the potential edges are missing. Since each atom has a fixed, small valence, the number of edges in a graph representing a molecule is no more than some fixed constant times n, the number of vertices. However, in an arbitrary graph the number of edges can be as large as $(n^2-n)/2$ (or larger, if there are multiple edges). An adjacency matrix for a sparse graph contains mostly zeros, but there is no good way of exploiting this fact. It has been proved that testing many graph properties, including isomorphism, requires examining some fixed fraction of the elements of the adjacency matrix in the worst case (6). Any algorithm which uses a matrix representation of a graph thus runs in time proportional to at least n^2 in the worst case. If we wish to deal with large graphs and hope to get a running time close to linear in the size of the graph, we must use a different representation.

The one we choose is an adjacency structure. An adjacency structure for a graph $G = (V,E)$ is a set of lists, one for each vertex. The list for vertex v contains all vertices adjacent to v. Note that a given edge (v,w) is represented twice; w appears in the adjacency list for v and v appears in the adjacency list for w. See Figure 3(c).

An adjacency structure is surprisingly easy to define and manipulate in FORTRAN or any other standard programming language. We use three arrays, which we may call adjacent to, vertex, and next. For any vertex v, the element $e_1 =$ adjacent to (v) represents the first element on the adjacency list for vertex v. The corresponding vertex is vertex(e_1), and the element



(a)

0	1	0	1	0	1
1	0	1	0	0	1
0	1	0	1	1	0
1	0	1	0	1	0
0	0	1	1	0	1
1	1	0	0	1	0

(b)

1: 2, 4, 6
2: 1, 3, 6
3: 2, 4, 5
4: 1, 3, 5
5: 3, 4, 6
6: 1, 2, 5

(c)

	1	2	3	4	5	6
adjacent to:	1	2	8	4	14	6

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
vertex:	2	1	4	1	6	1	3	2	6	2	4	3	5	3	5	4	6	5
next:	3	7	5	12		10	9	11		18	13	15		16		17		

(d)

Figure 3. Graphic representations: (a) graph, (b) adjacency matrix, (c) adjacency structure, and (d) array representation of adjacency structure

$e_2 = \underline{next}(e_1)$ represents the next element on the list. A null element indicates the end of the list. See Figure 3(d). The total amount of storage required by these arrays is $n+4m$, where n is the number of vertices in the graph and m is the number of edges; the total storage is thus linear in the size of the graph. Searches and other natural graph operations are easy to implement using such a data structure; e.g. see (7, 8). If the graph is labelled we can use two extra arrays which give vertex and edge labels. Athough the matrix representation of a graph is simple and mathematically elegant, the adjacency structure representation seems to be much more useful for computers.

3. Notions of Complexity.

If we are to discuss computer methods, we need some way of measuring the performance of an algorithm. We would like our code for molecules to be simple, natural, and easy to compute. Concepts like "simple" and "natural", although very important in any real-world cataloguing system, are difficult to define and quantify. We shall use a measure based on a machine's point of view, rather than on a human's. Though an algorithm good by such a measure may be unwieldy for human use, at best a method useful for machines will also be useful for people. At worst, such a measure provides a firm base for discussion of the merits of various methods.

One possible measure of algorithmic complexity is program size. Such a measure is related to the inherent simplicity or complexity of a method. This measure is static; it is independent of the size or structure of the particular input data. Some other possible measures are dynamic; they measure the amount of a resource used by the method as a function of the size of the input data. Typical dynamic measures are running time and storage space.

Program size as a measure has the disadvantage that in many cases the simplest algorithm is a brute force examination of all possibilities; the running time of such an algorithm is exponential in the size of the input and thus only very small graphs can be analyzed. The algorithms we shall consider all use storage space linear or quadratic in the number of vertices in the input graph; thus storage space as a measure does not discriminate finely enough for our purposes. The running time of an algorithm is strongly related to the algorithm's usefulness if it is run many times. We therefore choose running time as a function of input size as our measure of complexity.

How shall we measure running time? One possibility is to run the program several times on various sets of input data and extrapolate. This approach is very dangerous. If the number of examples tried is too small, the extrapolation is probably meaningless. If the number of examples tried is large and drawn

from a suitably defined random population, the extrapolation may be statistically meaningful. However, defining a random graph in a way which is realistic for chemistry is a very tricky problem. Furthermore any statistical method may miss rare but very bad cases; we would not like our cataloguing system to spend hours on an occasional bizarre molecule. We are therefore only satisfied with a careful theoretical analysis of an algorithm leading to a worst-case bound on its running time.

To account for variability in machines, we ignore constant factors and pay attention only to the asymptotic growth rate of the running time as a function of the size of the problem graph. Our measure is thus machine independent and most valid for large graphs. If machine-dependent constant factors and running time on small graphs are of interest, computer experiments or a more detailed analysis must be used. For convenience, we shall use the notation " $f(n)$ is $O(g(n))$ " to denote that the function $f(n)$ satisfies $f(n) \leq cg(n)$ for some positive constant c and all n, where f and g are non-negative functions of n.

4. Isomorphism and Subgraph Isomorphism.

The isomorphism problem for general graphs is not an easy one. Given two graphs G_1 and G_2 of n vertices, the number of possible one-to-one mappings of vertices is $n!$, and a brute force approach, which tries all the possibilities, is too time-consuming except for small graphs. A backtracking search (9), fares somewhat better. Initially, one vertex from each graph is chosen, and these vertices are matched. In general, some vertex w_1 adjacent to an already-matched vertex v_1 in G_1 is chosen and matched with some vertex w_2 adjacent to the vertex v_2 in G_2 previously matched to v_1. Then w_1 and w_2 are compared to make sure their adjacencies with already-matched vertices are consistent. If so, a new vertex for matching is chosen. If not, the last matched pair is unmatched and a new matching tried. The process continues until either all vertices are matched or there is found to be no way of matching the vertex sets of the two graphs.

Backtrack search saves time over the brute force method by abandoning an attempt at matching as soon as it is known to fail. The running time of backtrack search depends in a complicated way upon the structure of the graph; the best we can say in general is that if d is the maximum valence (number of vertices adjacent to a given vertex) in either graph, the maximum running time of backtrack search is $O((d-1)^n)$ -- still exponential, but better than brute force.

The most successful algorithms for general graph isomorphism use the backtrack approach (as a fall-back method) in combination

with a partitioning method (10,11,12,13). The idea is to partition the combined vertex sets of the two graphs so that any isomorphic mapping between the graphs preserves the partitioning. The method has four main steps.

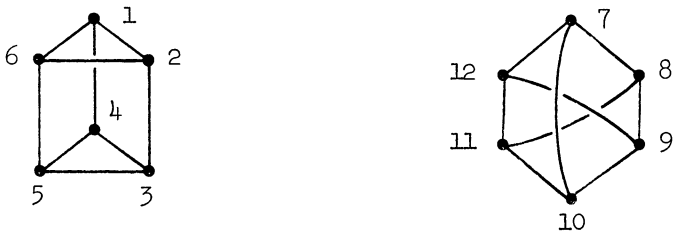
1. Choose an initial partition of the vertex sets.
2. Refine the partition. If any subset of the partition contains more vertices from one graph than from the other, go to step 4.
3. If each subset of the partition contains a single vertex from each graph, try the implied matching to see if it gives an isomorphism. If it does, halt with the isomorphism; if not, go to step 4. If some subset contains two or more vertices from one graph, choose a vertex in this subset from each graph, match these vertices, and go to step 2 (the new matching allows further refinement of the partition).
4. Backtrack. Back up to the partition existing when the last match was made. Try a new match and go to step 2. If all matches have been tried, back up to the previous match. If all possibilities for the very first match have been tried, halt. The graphs are not isomorphic.

For the initial partition we divide vertices up according to their labels and their valences. Other more elaborate partitionings are possible; see (14,15).

We carry out the refinement step in the following way. For each vertex, we determine the number of adjacent vertices in each subset of the partition. This information itself partitions the vertices. We take the intersection of this partition with the old partition as our new partition. We repeat this refining step until no further refinement takes place. Implementation of the repeated refinement step is somewhat tricky; Hopcroft (16) has provided a good implementation. The effect of matching two vertices in step 3 is to place them by themselves in a new subset of the partition. Thus step 3 guarantees refinement of the partition. See Figure 4 for an example of the application of the algorithm.

The idea behind this algorithm is to use all possible local means of distinguishing between vertices before guessing a match. The method seems to work quite well in practice. It is possible that some version of this partitioning method has a time bound which is a polynomial function of n. (To prove this requires showing that the amount of backtracking is polynomial in n; the refinement step requires only $O(m \log m)$ time, where m is the number of edges, if Hopcroft's implementation is used.) However, the present theoretical bounds on the algorithm are no better than those for backtrack search. It is a major open question whether a polynomial-time algorithm exists for the general graph isomorphism problem.

The situation for the subgraph isomorphism problem is somewhat better understood and somewhat more gloomy. It is possible



(a)

(b) {1,2,3,4,5,6,7,8,9,10,11,12}
A: valence 3

(c) {1,7} {2,3,4,5,6,8,9,10,11,12}
B C

(d) {1,7} {2,4,6,8,10,12}{3,5,9,11}
B D: 1B, 2C E: 3C

(e) {1,7} {2,6} {4,8,10,12} {3,5} {9,11}
B F: 1B, 1D, 1E G: 1B, 2E H: 2D, 1E I: 1B, 2D

Figure 4. Isomorphism test by partitioning: (a) graphs, (b) initial partition, (c) initial match 1–7, (d) first refinement, and (e) further refinement (match fails since F contains no vertices of second graph). Complete test requires matching 1 successively to 8, 9, 10, 11, 12, failing each time.

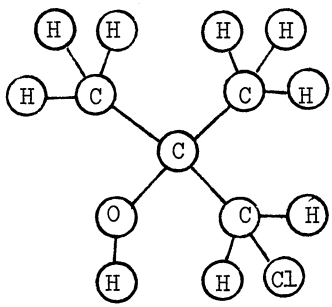


Figure 5. A tree

to generalize the partitioning algorithm described above so that it solves the subgraph isomorphism problem (17). However, the results of this method in practice seem to be mixed. Furthermore it has been proved that the subgraph isomorphism problem belongs to a class of problems called NP-complete. The NP-complete problems include a number of well-studied, apparently hard problems such as the travelling salesman problem of operations research, the tautology problem of propositional calculus, and many other combinatorial problems. The NP-complete problems have the property that if any one of them has a polynomial-time algorithm, they all do. Since no one has discovered a polynomial-time algorithm for any of these problems, though many people have tried, it seems likely that none of these problems is solvable in polynomial time. It is not known whether the graph isomorphism problem itself is NP-complete. For a discussion of NP-complete problems, see (18,19,20).

It would seem that our attempt to solve the graph isomorphism problem with a provably good algorithm is doomed to failure, and that we must be satisfied with a heuristic; that is, with a method which seems to work well in many cases for reasons which we do not understand. However, by lowering our sights somewhat, we can go a long way toward a solution which is both practical and theoretically efficient. We shall first consider the isomorphism problem for trees. For such graphs, there is a good isomorphism algorithm. Next, we study a decomposition method for representing a graph as a collection of smaller graphs joined in a tree-like fashion. We then examine the important special case of planar graphs. Finally, we combine these ideas to produce an isomorphism algorithm which is very fast on planar graphs and is likely to work well on most, if not all, chemical molecules.

5. Codes for Trees.

Let $G = (V,E)$ be a directed graph. A simple path from a vertex v_1 to a vertex v_k in G is a sequence of distinct edges $(v_1,v_2), (v_2,v_3), \ldots, (v_{k-1},v_k)$. The length of the path is $k-1$, the number of edges it contains. A cycle is a simple path from a vertex v_1 to itself. A graph is connected if every pair of vertices is joined by a path. In the description of a backtrack search in Section 4 we implicitly assumed that the graphs of interest were connected; we shall continue to make this assumption. A tree is a connected graph with no cycles (see Figure 5 for an example).

In contrast to the isomorphism problem for general graphs, the isomorphism problem for trees is relatively easy. Any tree with n vertices has exactly $n-1$ edges. We shall describe an algorithm for constructing, in $O(n)$ time, a code for any tree, such that two trees are isomorphic if and only if they have

identical codes. Variants of the algorithm have appeared in many places (21,22,23,24) and it has in fact been used in chemical computation (25).

To extract a unique code for a tree we must first put the tree into a canonical form. The first step in doing this is to find a uniquely determined vertex or edge in the tree. A tree has at least two vertices of valence one. We call such vertices leaves. For a given vertex v, let the height $h(v)$ of v be the length of the longest path from v to a leaf. A tree contains either a unique vertex of largest height, or two adjacent vertices of largest height (26). Since height must be preserved under isomorphism, this unique vertex or pair of vertices can be used as a starting point for construction of the canonical tree. If there are two vertices of largest height, we add a new vertex in the middle of the edge joining them and label it as a dummy vertex. Then we can assume our tree always has a unique vertex of largest height, which we call the root.

Each vertex v except the root has a unique parent u which is adjacent to v and satisfies $h(u) \geq h(v)+1$. All other vertices w adjacent to v are called its children and satisfy $h(w) \leq h(v)-1$. We define ancestors and descendants in the obvious way. Each vertex v in the tree defines a subtree consisting of v and its descendants (see Figure 6).

We define a total ordering with vertex labels by the following rules.

(1) If T and U are two trees with different labels on their roots, order the trees according to the labels of the roots.

(2) If T and U are two trees with the same label on their roots, let $T_1, T_2, \ldots, T_k$ be the subtrees defined by the children of the root of T (in increasing order) and let $U_1, U_2, \ldots, U_\ell$ be the subtrees defined by the children of the root of U. If there is some index j such that T_i is isomorphic to U_j for $i < j$ and T_j is less than U_j, or if T_i is isomorphic to U_i for $1 \leq i \leq k$ and $k < \ell$, then T is defined to be less than U.

That is, to compare two trees, we first compare their root labels. If these are identical, we order the subtrees defined by the children of the roots, and compare the ordered sequences of subtrees lexicographically.

Using this ordering, we can construct a canonical representation of a given tree by reordering the children of each vertex according to the order defined above. See Figure 6. From this canonical representation, we can construct a linear code which represents the tree uniquely. There are many possible ways to do this; one way is defined by the following rules.

(1) The code code(T) for a tree T consisting of a single vertex is its label.

(2) If T is a tree of more than one vertex, and $T_1, T_2, \ldots, T_k$ are the subtrees defined by the children of the roots of T (in order), then the code for T is code(T) = code(root)(code(T_1)code(T_2) ... code(T_k)) .

For instance, the code for the molecule in Figure 6 is C(C(ClHH)C(HHH)C(HHH)O(H)).

This method gives a unique code for each tree; two trees are isomorphic if and only if they have the same code (we have neglected to include edge labels in the code, but it is easy to do so if necessary). The code is quite natural, and it is easy to reconstruct a tree given its code. The reordering of subtrees is what guarantees that each tree has only one code. One can vary the exact definition of the ordering; what is important is that the subtrees be ordered somehow. When this algorithm is applied to chemical molecules, it is useful to use abbreviations in the code, such as omitting explicit reference to hydrogen atoms; e.g. see (27).

Implementing the reordering algorithm is somewhat complicated, since the sorting requires comparison of sequences element-by-element. See (28) for a good implementation. Constructing the code for a tree of n vertices requires O(n) time with this implementation. We can expect to find no faster algorithm, since any method must inspect the entire tree.

On trees, not only is the isomorphism problem efficiently solvable, but so is the subgraph isomorphism problem. Edmonds and Matula (29) have discovered an algorithm which will determine whether one tree is isomorphic to a subtree of another in $O(n^{5/2})$ time, where n is the number of vertices in the larger tree. This bound can be improved substantially if the valence of all vertices is bounded by a small constant. The algorithm may be of practical value, but this has yet to be tested.

6. Decomposition by Connectivity.

Though the algorithm of Section 5 for encoding trees is simple and fast, most chemical molecules are not trees. However, they are quite sparse and often tree-like. Our approach in this section will be to represent an arbitrary graph as a number of pieces linked in tree-like fashion. We can then encode the graph by encoding each piece separately, using these codes as labels on the linkage tree, and applying the tree encoding algorithm of Section 5 to encode the entire graph. In this way we can make the most out of our tree encoding method; the non-tree-like parts of the graph will usually be small.

To decompose a graph, we determine its connectivity. Let G = (V,E) be a connected graph. A cut set of G is a subset

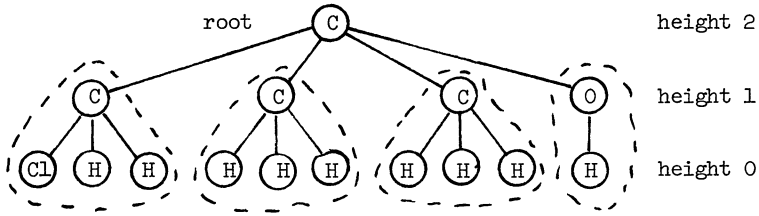


Figure 6. Tree of Figure 5 in canonical form. Dashes enclose subtrees of children of the root.

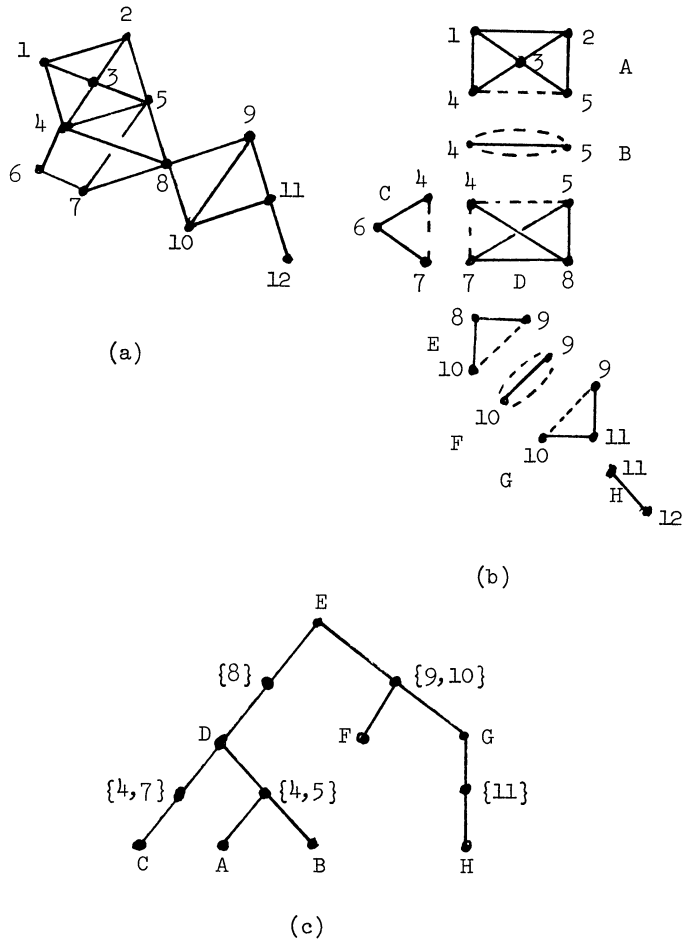


Figure 7. Schematic of (a) a graph, (b) its components, and (c) its decomposition tree

of vertices S such that there are at least two vertices v and w (not in S) for which every path from v to w passes through a vertex in S . Removal of the vertices in S thus breaks G into two or more connected pieces. If we add the vertices in G to each piece, the resultant subgraphs of G are called the components of G with respect to the cutset S .

We concentrate on cutsets containing no more than two vertices. By applying the following procedure, we break G into a number of smaller graphs.

Decomposition algorithm. Begin with a single component consisting of the entire graph. Repeat the following step until it no longer applies:

Find a cutset of size one or two in some component. If it is a cutset of size one, subdivide the component into its components with respect to the cutset. If it is a cutset of size two, say {v,w} , subdivide the component into its components with respect to the cutset, and add a new (dummy) edge (v,w) to each new component.

The importance for isomorphism testing of this algorithm is three-fold: first, the components found by the algorithm are essentially unique (preserved under isomorphism). (To guarantee uniqueness we must slightly modify the definition of components with respect to cutsets of size two; see (30,31,32). Second, the way the components fit together can be represented by a decomposition tree (33). This tree contains one vertex for each component and one vertex for each cutset. A cutset is adjacent to a component in the tree if the vertices of the cutset are in the component. Figure 7 gives an example of a graph, its components, and its decomposition tree.

Third, it is easy to find the components and the decomposition tree. An algorithm for this purpose, which uses depth first search (a systematic method of exploring a graph) has been developed (34,35,36). It runs in O(n+m) time on an n vertex, m edge graph.

Each component with respect to the decomposition is of one of three kinds -- a bond (single edge or set of multiple edges), a cycle, or a graph with no multiple edges and no cutsets of size one or two, called a triconnected graph. It is easy to encode bonds and cycles; all that is missing is a method of encoding triconnected graphs. If we can encode all the components, we can use the resultant codes as labels in the decomposition tree and apply the Section 5 algorithm to encode the entire tree. The running time of this algorithm will be O(n+m) for everything except the encoding of the triconnected components. If we use the partitioning method of Section 4 as a basis for encoding triconnected components, the complete algorithm will probably do quite well in practice. However, we have one more improvement to consider.

7. Planar Graphs.

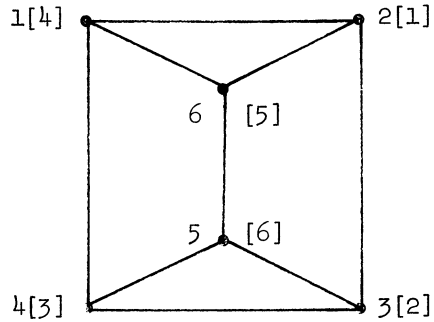
A planar graph is a graph which can be drawn on a piece of paper in such a way that no edges cross. Most chemical molecules (with the possible exception of complex organic molecules) are planar (note that this does not mean planar in the sense of stereochemistry). For planar graphs the isomorphism problem also has an easy solution.

When a graph is drawn in the plane, the drawing specifies a circular ordering of the edges around each vertex. A triconnected graph has the property that, if it is planar, its planar representation is unique up to mirror image. Thus there are only two ways of drawing a triconnected planar graph in the plane (two ways of specifying the circular ordering of edges around each vertex).

We can use this uniqueness to derive a code for any planar triconnected graph. First, we represent the graph in the plane. This can be done in $O(n)$ time (37). Next, we encode it. One way to do this was suggested by Weinberg (38). We explore the graph in the following way. We pick some starting edge and traverse it from one end to the other. When reaching the other end, we choose the next edge clockwise around the vertex and traverse it. We continue traversing edges in this way. Whenever we reach a vertex reached previously, we back up along the most recently traversed edge and pick the next edge clockwise. We continue the search until we have traversed each edge in both directions and returned to our starting point.

Such a search is uniquely determined by the choice of the starting edge and the direction to traverse it. We can construct a linear code during the search by writing a number (and a label) for each vertex reached, numbering the first vertex one, the next two, and so on. See Figure 8. To get a unique code, we construct a code for each possible edge and direction of traversal, for each of the two planar representations of the graph. Then we choose the lexicographically smallest of all the possible codes. A triconnected planar graph of $n \geq 3$ vertices has at most $3n-6$ edges (39), so we generate at most $12n-24$ codes, each of length n, and the total time to get a unique code is $O(n^2)$.

This encoding algorithm is very easy to program, but it is possible to get a faster algorithm by using more sophisticated methods. Hopcroft's partitioning algorithm (40) can be used to encode triconnected planar graphs in $O(n \log n)$ time (41). Hopcroft and Wong (42) have devised a very complicated algorithm which will encode a triconnected planar graph in $O(n)$ time. More recently, Fontet (43) has devised a simpler $O(n)$-time encoding algorithm. The practicality of these algorithms has yet to be tested.



(a)

(b) 1 2 3 4 1 4 5 6 1 6 2 6 5 3 5 4 3 2 1

(c) 1 2 3 4 1 4 5 1 5 6 2 6 3 6 5 4 3 2 1

Figure 8. (a) Planar Graph. (b) Code extracted by search starting with edge (1, 2). (Vertices are numbered in search order.) (c) Code extracted by search starting with edge (2, 3). (Numbers in brackets give the numbering for this search.) Code (c) is chosen since it is smaller lexicographically. All other codes are identical to either (b) or (c).

8. Summary and Other Applications.

We are now in a position to outline a complete isomorphism algorithm. We test isomorphism of two graphs by encoding each graph and testing the codes for equality. To encode a graph, we decompose it by finding all cutsets of size one and two, and forming the corresponding components and decomposition tree. We encode each bond component and each cycle component in some obvious way. We encode each triconnected component as follows. We test the component for planarity. If it is planar, we encode it using one of the methods in Section 7. If it is not planar, we encode it using the partitioning algorithm of Section 4. We use the codes for components as labels in the decomposition tree, and encode the tree (and thus the entire graph) using the method of Section 5.

The overall result is a method with a running time of $O(n+m)$ on n-vertex, m-edge graphs, plus whatever time is required to encode non-planar triconnected components. Though this algorithm has many parts, and programming it is quite a job, it has the potential to be of practical value. Though most of the parts of the algorithm have been programmed individually, the complete algorithm has not been programmed. Hopefully, this situation will be remedied in the near future.

Though the isomorphism problem is a formidable one, we have examined some ideas and some methods which can go a long way toward solving it. Many of the ideas we have considered have applications in other areas of chemistry. For instance, we have discussed representing a sparse graph as an adjacency matrix with many zeros. We can turn this idea around and use a graph to represent a sparse matrix (the matrix elements become labels for the corresponding graph edges). We can then apply graph-theoretic techniques to matrix problems such as solving a system of linear equations and computing eigenvalues and
A large literature has developed in this area; see (44,45,46) for instance.

Another application of graph theory to chemistry is in chromosome analysis. Suppose a chromosome is broken into a number of pieces and each piece analyzed. If this is done a number of times, the pieces found will overlap in various ways. The problem is to use the overlap information to reconstruct the entire chromosome. For linear chromosomes, a linear-time algorithm has been developed to solve this problem (47,48). For chromosomes which are rings, the problem seems surprisingly to be much harder and no good algorithm is known (49).

(1) "Survey of Chemical Notation Systems," National Academy of Sciences, National Research Council Publication 1150, 1964.
(2) Lederberg, J., "Dendral-64, a System for Computer Construction, Enumeration, and Notation of Organic Molecules as Tree Structures and Cyclic Graphs, Part I," NASA Scientific and Technical Aerospace Report, STAR No. N65-13158 and CR 57029, 1964.
(3) Lederberg, J., "Dendral-64, a System for Computer Construction, Enumeration and Notation of Organic Molecules as Tree Structures and Cyclic Graphs, Part II," NASA Scientific and Technical Aerospace Report, STAR No. N66-14074 and CR 68898, 1965.
(4) Sussenguth, E., Jr., J. Chem. Doc. (1965) 5, 36-43.
(5) Harary, F., "Graph Theory," 150-151, Addison-Wesley, Reading, Mass., 1969.
(6) Rivest, R. and Vuillemin, J., Seventh ACM Symp. on Theory of Computing (1975), 6-11.
(7) Hopcroft, J.E. and Tarjan, R. E., Comm. ACM (1973) 16, 372-378.
(8) Tarjan, R., SIAM J. Comput. (1972) 1, 146-160.
(9) Berztiss, A. T., Journal ACM (1973) 20, 365-377.
(10) Corneil, D. G. and Gotlieb, C. C., Journal ACM (1970) 17, 51-64.
(11) Schmidt, D. C. and Druffel, L. E., Journal ACM (1976) 23, 433-445.
(12) Sussenguth, E., Jr., J. Chem. Doc. (1965) 5, 36-43.
(13) Unger, S. H., Comm. ACM (1964) 7, 26-34.
(14) Corneil, D. G. and Gotlieb, C. C., Journal ACM (1970) 17, 51-64.
(15) Schmidt, D. C. and Druffel, L. E. Journal ACM (1976) 23, 433-445.
(16) Hopcroft, J. E., in Kohavi, Z. and Paz, A., eds., "Theory of Machines and Computations," 189-196, Academic Press, New York, 1971.
(17) Sussenguth, E., Jr., J. Chem. Doc. (1965) 5, 36-43.
(18) Cook, S., Third ACM Symp. on Theory of Computing (1971), 151-158.
(19) Karp, R. M., in Miller, R. E. and Thatcher, J. W., eds., "Complexity of Computer Computations," 85-104, Plenum Press, New York, 1972.
(20) Karp, R. M., Networks (1975) 5, 45-68.
(21) Busacker, R. G. and Saaty, T. L., "Finite Graphs and Networks: An Introduction with Applications," 196-199, McGraw-Hill, New York, 1965.
(22) Lederberg, J., NASA Scientific and Technical Aerospace Report, STAR No. N65-13158 and CR 57029, 1964.
(23) Scoins, H. I., Machine Intelligence (1968) 3, 43-60.
(24) Weinberg, L., Proc. Third Annual Allerton Conf. on Circuit and System Theory (1965), 733-744.

(25) Lederberg, J., NASA Scientific and Technical Aerospace Report, STAR No. N65-13158 and CR 57029, 1964.
(26) Harary, F., "Graph Theory," 35-36, Addison-Wesley, Reading, Mass., 1969.
(27) Lederberg, J., NASA Scientific and Technical Aerospace Report, STAR No. N65-13158 and CR 57029, 1964.
(28) Aho, A. V., Hopcroft, J. E., and Ullman, J. D., "The Design and Analysis of Computer Algorithms," 84-86, Addison-Wesley, Reading, Mass., 1974.
(29) Matula, D. W., SIAM Review (1968) 10, 273-274.
(30) Hopcroft, J. E. and Tarjan, R. E., SIAM J. Comput. (1973) 2, 135-158.
(31) Maclaine, S., Duke Math. J. (1937) 3, 460-472.
(32) Tutte, W. T., "Connectivity in Graphs," University of Toronto Press, Toronto, 1966.
(33) Harary, F., "Graph Theory," 36-37, Addison-Wesley, Reading, Mass., 1969.
(34) Hopcroft, J. E. and Tarjan, R. E., SIAM J. Comput. (1973) 2, 135-158.
(35) Hopcroft, J. E. and Tarjan, R. E., Comm. ACM (1973) 16, 372-378.
(36) Tarjan, R. E., SIAM J. Comput. (1972) 1, 146-160.
(37) Hopcroft, J. E. and Tarjan, R. E., Journal ACM (1974) 21, 549-568.
(38) Weinberg, L., IEEE Trans. on Circuit Theory (1966) CT-13, 142-148.
(39) Harary, F., "Graph Theory," 104, Addison-Wesley, Reading, Mass., 1969.
(40) Hopcroft, J. E., in Kohavi, Z. and Paz, A., eds., "Theory of Machines and Computations," 189-196. Academic Press, New York, 1971.
(41) Hopcroft, J. E. and Tarjan, R. E., Journal of Computer and System Sciences (1973) 7, 323-331.
(42) Hopcroft, J. E. and Wong, J. K., Sixth Annual ACM Symp. on Theory of Computing (1973), 172-184.
(43) Fontet, M., Proc. Third International Colloquium on Automata, Languages, and Programming, to appear.
(44) Bunch, J. R. and Rose, D. J., eds., "Sparse Matrix Computations," Academic Press, New York, 1976.
(45) Duff, I. S., "A Survey of Sparse Matrix Research," Technical Report CSS 528, Computer Science and Systems Division, AERE Harwell, 1976.
(46) Rose, D. J. and Willoughby, R., eds., "Sparse Matrices and their Applications," Plenum Press, New York, 1972.
(47) Benzer, S., Proc. of the National Academy of Sciences (1959) 45, 1607-1620.
(48) Lueker, G. S. and Booth, K. S., Seventh ACM Symp. on Theory of Computing (1975), 255-265.
(49) Booth, K. S., "P-Q Trees," Ph.D. Thesis, Dept. of Electrical Engineering and Computer Sciences, University of California, Berkeley, 1975.