Algorithms for Chemical Computations

Ralph E. Christoffersen

ACS Symposium Series 46

Algorithms for Chemical Computations

Ralph E. Christoffersen, EDITOR
The University of Kansas

A symposium sponsored by the Division of Computers in Chemistry at the 171st Meeting of the American Chemical Society, New York, N.Y., Aug. 3[illegible]6.

ACS SYMPOSIUM SERIES **46**

AMERICAN CHEMICAL SOCIETY
WASHINGTON, D. C. 1977

504/25

Library of Congress CIP Data

Algorithms for chemical computations.
(ACS symposium series; 46 ISSN 0097-6156)

Includes bibliographical references and index.

1. Chemistry—Data processing—Congresses. 2. Algorithms—Congresses.

I. Christoffersen, Ralph E., 1937- . II. American Chemical Society. Division of Computers in Chemistry. III. Series: American Chemical Society. ACS symposium series; 46.

QD39.3.E46A43 540'.28'5 77-5030
ISBN 0-8412-0371-7 ACSMC8 46 1-151

FOREWORD

The ACS SYMPOSIUM SERIES was founded in 1974 to provide a medium for publishing symposia quickly in book form. The format of the SERIES parallels that of the continuing ADVANCES IN CHEMISTRY SERIES except that in order to save time the papers are not typeset but are reproduced as they are submitted by the authors in camera-ready form. As a further means of saving time, the papers are not edited or reviewed except by the symposium chairman, who becomes editor of the book. Papers published in the ACS SYMPOSIUM SERIES are original contributions not published elsewhere in whole or major part and include reports of research as well as reviews since symposia may embrace both types of presentation.

PREFACE

As computing hardware and software continues to pervade the various areas of chemical research, education, and technology, various important developments begin to emerge. For example, for areas in which large "number crunching" is required, larger and faster computing systems have been developed that incorporate parallel processing, which have provided substantial increases in speed of problem solving compared with sequential processing. In other areas, such as data acquisition and equipment control, minicomputers and "midicomputers" have been designed and built to provide substantial improvements in both the quality of the data collected and the implementation of new experiments that could not be performed without the computer system assistance. Equally important developments in software have also evolved, from the implementation of convenient timesharing systems for program development to the development of a variety of application program "packages" for use in various chemical research areas.

While the limits achievable through better hardware design or more efficient programming of available algorithms are far from being reached, it is now becoming apparent that the algorithms themselves may present both substantial difficulties and opportunities for significant progress. In other words, it may no longer be a feasible strategy to assume that either a faster computer or a more efficiently programmed existing algorithm will be adequate in solving a given problem.

To focus more clearly on this emerging area of importance, a symposium was organized as a part of the Fall American Chemical Society Meeting in San Francisco, on August 30, 1976. The goal was to bring together several experts in the development of algorithms for chemical research so that the state of the art might be assessed. These persons, whose papers are included in this volume, discussed not only the significant developments in algorithms that have already occurred, but also indicated places where currently available algorithms were not adequate.

While it is not possible in a single symposium to discuss the entire spectrum of areas where significant algorithmic development has occurred or is needed, an attempt was made to include several of the important areas where progress is evident. In particular, the papers in this volume include discussions of the use of graph theory in algorithm design, algorithm design and choice in quantum chemistry, molecular scattering, solid state description and pattern recognition, and the handling of chemical information. As both the authors and the topics indicate, the general topic is extremely diverse in scope, involving expertise from several disciplines in the search for new and improved algorithms. While this area is currently in its infancy, its potential impact is great, and it is hoped that these papers will serve both as a reference to the current state of the art and as an impetus to extend the study of algorithmic development to other areas as well.

The University of Kansas
Lawrence, Kansas
December 1976

RALPH E. CHRISTOFFERSEN

CONTENTS

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.

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

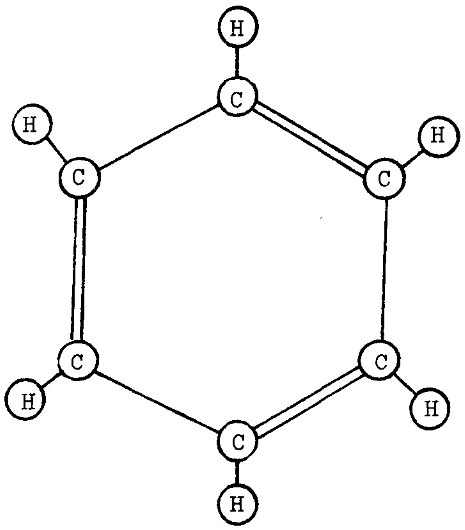


Figure 1. Graphic representation of benzene

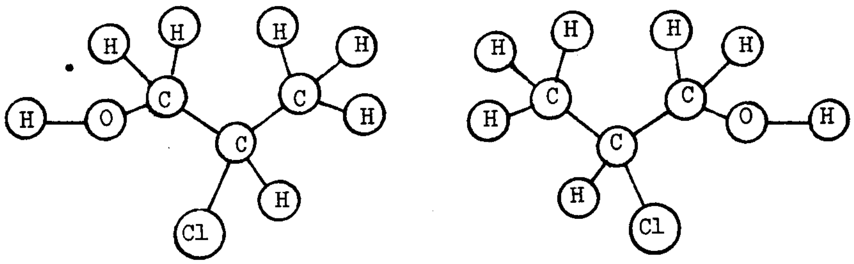


Figure 2. Stereoisomers

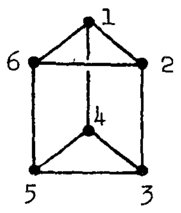
two standard ways to represent graphs in a computer. The first is by an adjacency matrix. If $G = (V,E)$ is a graph with n vertices numbered from 1 to n, an adjacency matrix for G is the n by n matrix $M = (m_{ij})$ with elements 0 and 1, such that $m_{ij} = 1$ if (v_i,v_j) is an edge of G and $m_{ij} = 0$ 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)

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

(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
<u>adjacent to</u>:	1	2	8	4	14	6

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