An Algorithm for Subgraph Isomorphism

J. R. ULLMANN

National Physical Laboratory, Teddington, Middlesex, England

ABSTRACT. Subgraph isomorphism can be determined by means of a brute-force tree-search enumeration procedure. In this paper a new algorithm is introduced that attains efficiency by inferentially eliminating successor nodes in the tree search. To assess the time actually taken by the new algorithm, subgraph isomorphism, clique detection, graph isomorphism, and directed graph isomorphism experiments have been carried out with random and with various nonrandom graphs.

A parallel asynchronous logic-in-memory implementation of a vital part of the algorithm is also described, although this hardware has not actually been built The hardware implementation would allow very rapid determination of isomorphism.

KEY WORDS AND PHRASES. graph, graph isomorphism, directed graph isomorphism, digraph isomorphism, subgraph, subgraph isomorphism, clique, clique detection, isomorphism algorithm, tree search, search tree, game tree, parallel processing, array processing, special purpose computer, logic-in-memory arrays, asynchronous sequential circuits, Boolean matrices

CR CATEGORIES: 3 64, 5 32, 6 22

1. *Introduction*

Corneil and Gotlieb [4] mention that one of the possible applications of subgraph isomorphism is for finding whether a given chemical compound is a subcompound of a further specified compound, given the structural formulas. Subgraph isomorphism may be useful in scene analysis [1, 10] for detecting a relationally described object that is embedded in a scene. Problems akin to subgraph isomorphism have also arisen in research on the recognition of distorted shapes, where any admissible distortion conserves positional relationships within limits. There is some formal similarity between the problems of finding whether two graphs are related by a 1:1 correspondence that conserves adjacency and finding whether two patterns are related by a distortion that conserves spatial relationships within known limits. This idea is explored at an introductory level in [11, Sec. 7.3]; [11] also indicates the historical origin of the algorithm that is described in the present paper.

It is well known that isomorphism can be determined by brute-force enumeration. As a first step toward introducing the original part of our algorithm, Section 2 of this paper describes a brute-force enumeration procedure that is actually a depth-first tree-search algorithm. Section 3 introduces the original part of the work, which consists of a procedure that is entered after each node in the tree search. The result of this procedure is generally a reduction in the number of successor nodes that must be searched, which yields a reduction in the total computer time required for determining isomorphism.

In Corneil and Gotlieb's algorithm [4], the two graphs that are to be tested for isomorphism are separately subjected to a computation which produces representative

Author's address. Division of Computer Science, National Physical Laboratory, Teddington, Middlesex TW11 OLW, England.

graphs. Like the tree-search algorithm of Berztiss [2], which we will discuss later, our algorithm differs from that of Corneil and Gotlieb in that it does not process the two graphs separately: the ability to cope with subgraph isomorphism stems from the fact that the computation always works on both graphs at once. We have not attempted to find a graph property that is possessed by all graphs which are isomorphic to a given graph.

In this paper we will use the terminology of Nilsson [7] for tree-search computations, and we will use the terminology of Harary [5] for graphs other than search trees. A *graph* G consists of a finite nonempty set V of p elements that are called *points*, together with a set E of q distinct unordered pairs of distinct points that belong to V. A pair of points that belongs to E is a *line*. A *subgraph* of G is a graph whose points and lines all belong to G. A graph G_α is *isomorphic* to a subgraph of a graph G_β if and only if there is a 1 : 1 correspondence between the point sets of this subgraph and of G_α that preserves adjacency.

2. *Simple Enumeration Algorithm for Subgraph Isomorphism*

In this section we formulate a simple tree-search algorithm; for introductory purposes we omit the vital procedure that eliminates successor nodes in the search. This procedure is introduced in Section 3.

The enumeration algorithm is designed to find all of the isomorphisms between a given graph $G_\alpha = (V_\alpha, E_\alpha)$ and subgraphs of a further given graph $G_\beta = (V_\beta, E_\beta)$. The numbers of points and lines of G_α and G_β are p_α, q_α and p_β, q_β, respectively. The adjacency matrices of G_α and G_β are $A = [a_{i,j}]$ and $B = [b_{i,j}]$, respectively.

For reasons that will soon become apparent, we define an M' matrix to be a p_α (rows) $\times$ p_β (columns) matrix whose elements are 1's and 0's, such that each row contains exactly one 1 and no column contains more than one 1. A matrix $M' = [m'_{i,j}]$ can be used to permute the rows and columns of B to produce a further matrix C. Specifically, we define $C = [c_{i,j}] = M'(M'B)^{\mathrm{T}}$, where T denotes transposition. If it is true that

$$(\forall i \; \forall j)_{\substack{1 \le i \le p_\alpha \\ 1 \le j \le p_\alpha}} (a_{i,j} = 1) \Rightarrow (c_{i,j} = 1), \tag{1}$$

then M' specifies an isomorphism between G_α and a subgraph of G_β. In this case, if $m'_{i,j} = 1$, then the jth point in G_β corresponds to the ith point in G_α in this isomorphism.

At the start of the enumeration algorithm, we construct a $p_\alpha \times p_\beta$ element matrix $M^0 = [m^0_{i,j}]$ in accordance with

$$m^0_{i,j} \begin{cases} = 1 \text{ if the degree of the } j\text{th point of } G_\beta \text{ is greater than or equal to the degree of} \\ \qquad \text{the } i\text{th point of } G_\alpha, \\ = 0 \text{ otherwise.} \end{cases}$$

Indeed we would set $m_{i,j} = 0$ if we had any a priori reason to be sure that the jth point of G_β could not correspond to the ith point of G_α in any subgraph isomorphism.

The enumeration algorithm works by generating all possible matrices M' such that for each and every element $m'_{i,j}$ of M', $(m'_{j,i} = 1) \Rightarrow (m^0_{j,i} = 1)$. For each such matrix M' the algorithm tests for isomorphism by applying condition (1). Matrices M' are generated by systematically changing to 0 all but one of the 1's in each of the rows of M^0, subject to the definitory condition that no column of a matrix M' may contain more than one 1. In the search tree, the terminal nodes are at depth $d = p_\alpha$ and they correspond to distinct matrices M'. Each nonterminal node at depth $d < p_\alpha$ corresponds to a distinct matrix M which differs from M^0 in that in d of the rows, all but one of the 1's has been changed to 0.

The algorithm uses a p_β-bit binary vector $\{F_1, \cdots, F_i, \cdots, F_{p_\beta}\}$ to record which columns have been used at an intermediate state of the computation. $F_i = 1$ if the ith column has been used. The algorithm also uses a vector $\{H_1, \cdots, H_d, \cdots, H_{p_\alpha}\}$ to

record which column has been selected at which depth. $H_d = k$ if the kth column has been selected at depth d.

We shall use the symbol := to denote assignment. Thus $d := d + 1$ means "set d equal to $d + 1$." Further, we shall write $M := M_d$, meaning "set the entire matrix M equal to matrix M_d." Matrix M_d is a stored copy of matrix M at depth d. We have formulated the algorithm so that it is as similar as possible to the algorithm of Section 3 (for instance the complete assignment $M_d := M$ is not really necessary in the present section).

The simple enumeration algorithm is as follows.

Step 1 $M = M^0$, $d := 1$; $H_1 = 0$;
for all $i = 1, \cdots, p_\alpha$, set $F_i := 0$;
Step 2 If there is no value of j such that $m_{dj} = 1$ and $F_j = 0$ then go to step 7;
$M_d = M$,
if $d = 1$ then $k := H_1$ else $k := 0$,
Step 3 $k := k + 1$,
if $m_{dk} = 0$ or $F_k = 1$ then go to step 3;
for all $j \neq k$ set $m_{dj} := 0$,
Step 4. If $d < p_\alpha$ then go to step 6 else use condition (1) and give output if an isomorphism is found;
Step 5 If there is no $j > k$ such that $m_{dj} = 1$ and $F_j = 0$ then go to step 7;
$M := M_d$,
go to step 3;
Step 6 $H_d = k$, $F_k = 1$; $d = d + 1$;
go to step 2,
Step 7 If $d = 1$ then terminate algorithm,
$F_k = 0$; $d := d - 1$, $M = M_d$, $k := H_d$,
go to step 5,

3. *Algorithm Employing Refinement Procedure*

To reduce the amount of computation required for finding subgraph isomorphisms we employ a procedure, which we call the *refinement procedure*, that eliminates some of the 1's from the matrices M, thus eliminating successor nodes in the tree search.

To introduce the refinement procedure, let us consider the matrix M that is associated with any given nonterminal node in the search tree. Any subgraph isomorphism corresponds to a particular matrix M'. We say that an isomorphism is an isomorphism *under* M if its terminal node in the search tree is a successor of the node with which M is associated. The 0's in the matrix M merely preclude correspondences between points of V_α and V_β. If $m'_{ij} = 0$ for all isomorphisms under M, then if $m_{ij} = 1$ we can change $m_{ij} = 1$ to $m_{ij} = 0$ without losing any of the isomorphisms under M: all such isomorphisms will still be found by the tree search. In the next paragraph we work out a condition that is necessarily satisfied if $m'_{ij} = 1$ for any isomorphisms under M. If this necessary condition is not satisfied and $m_{ij} = 1$, then the refinement procedure changes $m_{ij} = 1$ to $m_{ij} = 0$.

Let $v_{\alpha i}$ be the ith point in V_α, and let $v_{\beta j}$ be the jth point in V_β. Let $\{v_{\alpha 1}, \cdots, v_{\alpha x}, \cdots, v_{\alpha\gamma}\}$ be the set of all points of G_α that are adjacent to $v_{\alpha i}$ in G_α. Let us consider the matrix M' that is associated with any given isomorphism under M. From the definition of subgraph isomorphism it is necessary that if $v_{\alpha i}$ corresponds to $v_{\beta j}$ in the isomorphism, then for each $x = 1, \cdots, \gamma$ there must exist a point $v_{\beta y}$ in V_β that is adjacent to $v_{\beta j}$, such that $v_{\beta y}$ corresponds to $v_{\alpha x}$ in the isomorphism. If $v_{\beta y}$ corresponds to $v_{\alpha x}$ in the isomorphism, then the element of M' that corresponds to $\{v_{\alpha x}, v_{\beta y}\}$ is 1. Therefore if $v_{\alpha i}$ corresponds to $v_{\beta j}$ in *any* isomorphism under M, then for each $x = 1, \cdots, \gamma$ there must be a 1 in M corresponding to some $\{v_{\alpha x}, v_{\beta y}\}$ such that $v_{\beta y}$ is adjacent to $v_{\beta j}$. In other words, if $v_{\alpha i}$ corresponds to $v_{\beta j}$ in any isomorphism under M, then

$$\underset{1 \leq x \leq p_\alpha}{(\forall x)} \left((a_{ix} = 1) \Rightarrow \underset{1 \leq y \leq p_\beta}{(\exists y)} \left(m_{xy} \cdot b_{yj} = 1\right)\right). \tag{2}$$

The refinement procedure simply tests each 1 in M to find whether condition (2) is satisfied. For any $m_{ij} = 1$ such that (2) is not satisfied, $m_{ij} = 1$ is changed to $m_{ij} = 0$. Such changes may cause condition (2) to be no longer satisfied for further 1's in M, so that further changes can be made, and so on. In fact the refinement procedure applies condition (2) in turn to each 1 in M, and it then does this over and over again until there is an iteration in which all the 1's in M are processed and none of them is changed to 0. Note that there is no restriction on the order in which the 1's in M should be processed; this means that the refinement procedure can be implemented in asynchronous hardware (see Section 5).

Generally the result of the refinement procedure is to change some of the 1's in M to 0's. However, the refinement procedure may leave M unchanged, and this is particularly important when M is a matrix M'. A necessary and sufficient condition for subgraph isomorphism is that the refinement procedure leaves M' unchanged. This follows because if M' is unchanged by the refinement procedure, then (2) holds for each 1 in M'. Therefore M' specifies a 1:1 mapping of V_α into V_β such that if two points are adjacent in G_α then the two corresponding points in G_β are adjacent. We can therefore use the refinement procedure as a test for subgraph isomorphism instead of using condition (1): if the refinement procedure results in any 1 in M' being changed to 0, then M' does not specify an isomorphism.

During the refinement procedure we continually check whether any row of M contains no 1. If any row of M contains no 1 then the procedure jumps to its FAIL exit, because there is no advantage in continuing the procedure. Otherwise the procedure terminates at its SUCCEED exit.

In the detailed program implementation, we do not use one computer word per element of A, B, and M. Instead we ensure that each row of M is contained in a separate computer word and each column of B is contained in a separate computer word. To implement condition (2), we *and* the word containing the xth row of M with the word containing the jth column of B, and test whether the resulting word contains any 1's. This is, of course, much faster than bit-by-bit computation, and it is important that the refinement procedure can in this way exploit the limited parallelism of an ordinary digital computer. The refinement procedure is formulated in Appendix 1.

Using the refinement procedure, our algorithm for subgraph isomorphism is as follows:

Step 1 $M = M^0$, $d = 1$, $H_1 = 0$,
for all $i = 1, \cdots, p_\alpha$ set $F_i := 0$;
refine M, if exit FAIL then terminate algorithm;
Step 2. If there is no value of j such that $m_{dj} = 1$ and $f_j = 0$ then go to step 7,
$M_d := M$;
if $d = 1$ then $k = H_1$ else $k := 0$;
Step 3 $k = k + 1$,
if $m_{dk} = 0$ or $f_k = 1$ then go to step 3;
for all $j \neq k$ set $m_{dj} = 0$;
refine M; if exit FAIL then go to step 5;
Step 4 If $d < p_\alpha$ then go to step 6 else give output to indicate that an isomorphism has been found;
Step 5 If there is no $j > k$ such that $m_{dj} = 1$ and $f_j = 0$ then go to step 7,
$M = M_d$;
go to step 3,
Step 6 $H_d = k$, $F_k = 1$; $d = d + 1$;
go to step 2,
Step 7. If $d = 1$ then terminate algorithm,
$F_k = 0$; $d = d - 1$; $M := M_d$, $k := H_d$;
go to step 5;

For simplicity we have formulated the algorithm so that $d = 1, 2, \cdots, p_\alpha$ correspond respectively to the 1st, 2nd, $\cdots, p_\alpha$th rows of adjacency matrix A, but we have not followed this in our experiments. Instead we have arranged that $d = 1, 2, \cdots, p_\alpha$ correspond respectively to the points of G_α in order of decreasing degree. This is intended

to enhance the effect of the refinement procedure at nodes near to the root node of the search tree, since a point of high degree is adjacent to more points than a point of low degree. An alternative strategy might be more appropriate in a specific application of the algorithm.

The refinement procedure necessarily converges in a finite number of steps because it never changes a 0 to a 1 in M and the number of 1's in M is finite. Our complete algorithm for subgraph isomorphism is truly an algorithm: it necessarily finds all subgraph isomorphisms within a finite time. To assess the time actually taken by the algorithm, we have resorted to experiment.

4. *Experiments*

Experiments were carried out with a KDF9 computer, which is a somewhat unusual machine of approximately 1963 vintage. This machine does logic and arithmetic on the last-in words of a last-in-first-out stack of words. To add together the two last-in words takes 1 μsec, and to fetch a 48-bit word from the core store to the stack takes 9 μsec. These figures are mentioned here in order to endow our computer-time results with a little (but unfortunately not more than a little) meaning. The programs were written in the assembly language of the KDF9.

We used a pseudorandom number generator [9] to construct adjacency matrices. The program was written so that the probability of an off-diagonal element being 1 was approximately 0.25. Each adjacency matrix produced by this program was tested for connectedness, and if the corresponding graph was found not to be connected,[1] then the adjacency matrix was rejected and a new one was constructed using further pseudorandom numbers. In our experiments, all graphs were generated in this way, unless otherwise stated. Figure 1 shows q versus p for such graphs. Each cross in Figure 1 denotes an average value of q over 50 trials with different graphs.

In our statements of experimental results, s.d. always means the square root of $(1/n)\sum_1^n z^2 - ((1/n)\sum_1^n z)^2$ when there are n trials with variate z. Although in this work the distributions are generally very skewed, we give s.d. values as a better-than-nothing rough measure of the variability of the variate. In Figures 1 and 2, the length of the vertical line through a cross is twice the s.d. value. Every random-graph result that is reported below was obtained over fifty trials with different graphs.

The storage requirements of our algorithm are small except for the storage of p_α matrices M, which occupy $p_\alpha{}^2 p_\beta$ bits, or $p_\alpha{}^2$ words in our implementation.

SUBGRAPH ISOMORPHISM. For selected values of p_α and p_β such that $p_\alpha < p_\beta$, adjacency matrices A and B were generated as described above. Matrix A was *or*'ed into matrix B by means of the following procedure: For each $i, j = 1, \cdots, p_\alpha$ set $b_{i,j} := b_{i,j} \vee a_{i,j}$. Of course the resulting B matrices had higher values of q_β than those indicated in Figure 1. For each pair of matrices A and B, we applied the subgraph isomorphism algorithm of Section 3; Table I summarizes the results.

CLIQUE DETECTION. A *clique* is a maximal complete subgraph [5]. As a further demonstration of the subgraph isomorphism algorithm we applied it, after suitable modification, to the detection of cliques. We used the obvious method in which the subgraph isomorphism algorithm is applied to G_β and complete graphs G_α for successively smaller values of p_α until at least one isomorphism is found. The modifications to the Section 3 algorithm, and the reasons for them, are given in Appendix 2. Experimental results with random graphs are summarized in Table II.

In the complete 3-partite graph $K(3, 3, 3)$ our program found 27 3-point cliques in

[1] If G_α was a connected graph and if G_β was a disconnected graph consisting of, for example, two connected subgraphs $G_{\beta 1}$ and $G_{\beta 2}$, then we could test for isomorphisms between G_α and subgraphs of $G_{\beta 1}$ and $G_{\beta 2}$ separately, thus reducing the amount of computation required. Although this would be helpful in practice, it would tend to complicate our experiments, and this is why we have experimented only with connected graphs.

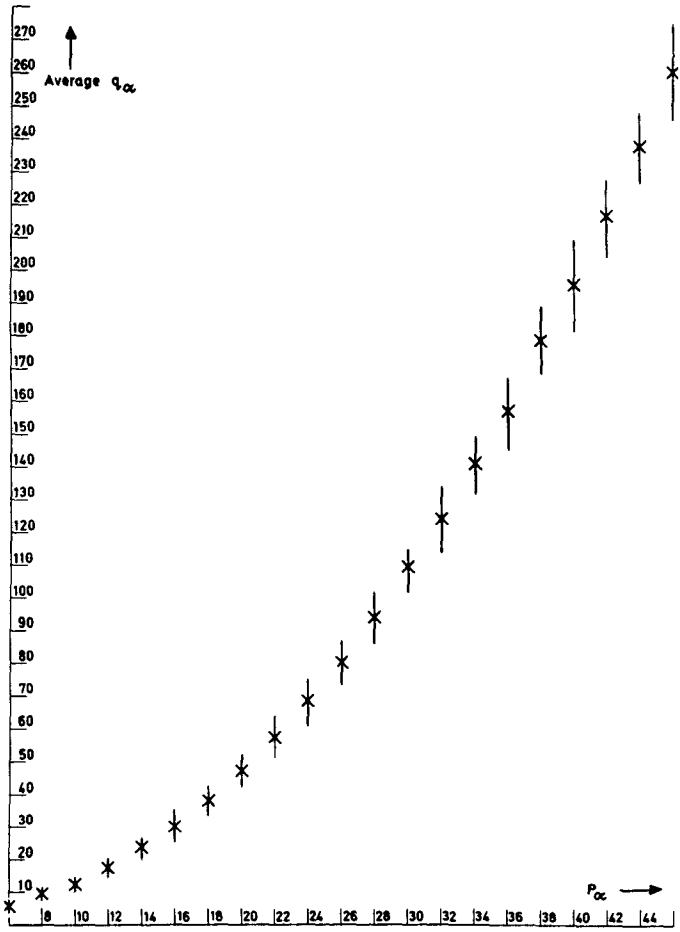

FIG 1. Number of lines versus number of points for pseudorandomly generated graphs

TABLE I. RESULTS OF EXPERIMENTS WITH SUBGRAPH ISOMORPHISM

p_α	p_β	q_β		Number of isomorphisms		Time in seconds	
		av.	s.d.	av.	s.d	av	s.d.
6	12	21.1	3.1	960 8	140.4	14.5	13.11
8	12	23.5	2 9	1223.0	142.8	44.5	55 4
10	12	26.0	3.4	949 1	121 2	124.0	90 4
7	14	28.3	4.0	4769 9	88.9	97.6	118 7

TABLE II. RESULTS OF EXPERIMENTS WITH CLIQUE DETECTION

p_β	Points per clique		Number of cliques		Time in seconds	
	av.	s.d.	av.	s.d.	av.	s.d.
12	3.1	0.3	2.9	1 8	0 3	0 5
16	3.2	0.4	5.3	3 5	0.7	0.5
20	3.5	0.5	7.1	6.6	1.6	0 6
24	3.8	0.5	6.4	8.3	3 1	0.8
28	3.9	0.4	7.0	9.7	6.3	1.6

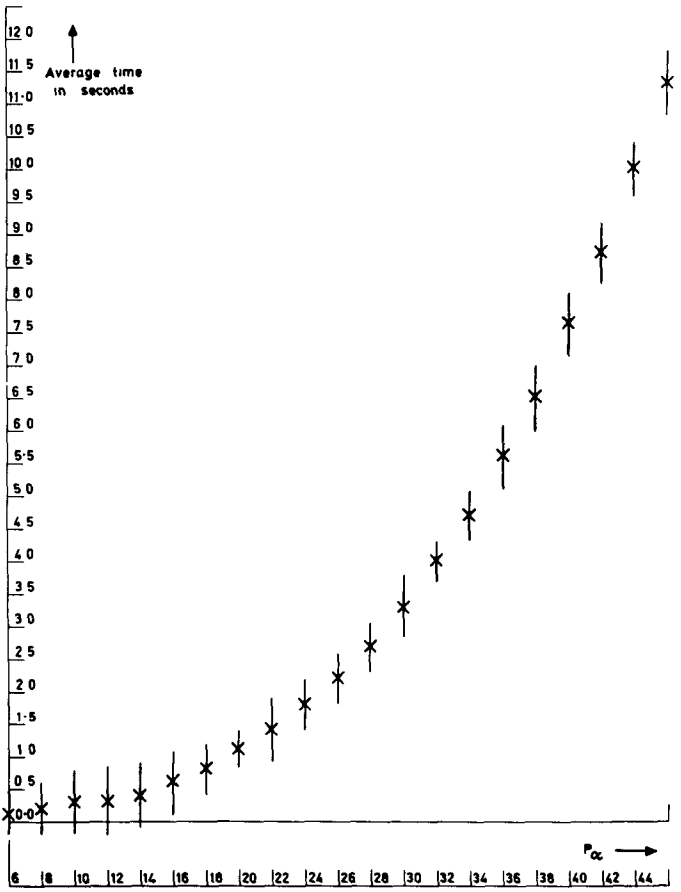

FIG. 2. Time in seconds versus number of points for determination of isomorphism of pseudo-randomly generated graphs

0.6 sec, and in the complete 4-partite graph $K(3, 3, 3, 3)$ our program found 81 4-point cliques in 3.2 sec. For these graphs the tree-search clique detection procedure of Bron and Kerbosch [3] appears to be faster than ours, bearing in mind that their procedure was written in ALGOL, ours was written in assembly language, and the KDF9 and EL-X8 are similar in speed. Osteen and Tou [8] have also reported that their clique detection algorithm found these cliques in less time than ours, but using an IBM 360/65, which is very roughly three times as fast as a KDF9.

GRAPH ISOMORPHISM. Figure 2 shows computing time versus p_α for determining all isomorphisms between two identical graphs G_A and $G_B = G_A$. Here the matrices M^0 were constructed according to

$$m^0_{i,j} = \begin{cases} 1 \text{ if the degree of the } i\text{th point of } G_A \text{ is the same as the degree of the} \\ \quad j\text{th point of } G_B, \\ 0 \text{ otherwise.} \end{cases} \tag{3}$$

For $p_\alpha > 20$ there was never more than one isomorphism between G_A and G_B.

For $p_\alpha = 6, 8, 10$ we also determined all isomorphisms between G_A and $G_B = G_A$ using the simple enumeration algorithm of Section 2, with M^0 constructed according to (3). In over 50 trials the average times for $p_\alpha = 6, 8, 10$ were 0.2 sec, 1.1 sec, and 13.74 sec, respectively. Comparing these results with Figure 2, we see that on the average the algorithm of Section 3 finds all isomorphisms between a pair of 46-point 260-line graphs

more quickly than the algorithm of Section 2 finds all isomorphisms between a pair of 10-point 13-line graphs.

From Figure 2 we see that for an average edge density equal to 0.25, the timing of our algorithm depends roughly on p_α^3, whereas Corneil and Gotlieb [4] have reported that the timing of their algorithm on isomorphic random graphs depends on p_α^2. They have specifically reported that their algorithm took 0.00447 min on an IBM 7094-II for edge density = 0.5 and $p_\alpha = 20$. Our algorithm took 0.0217 min = 1.3 sec on the average for these graphs, and it took 0.9 sec when we used the faster version of the refinement procedure that is mentioned at the end of Appendix 1. For isomorphic random graphs, the Corneil and Gotlieb algorithm appears to be more efficient than ours.

A referee commented that even a poor algorithm for isomorphism may work quite well with random graphs. To provide a more stringent test, the referee kindly provided, with the permission of D. Corneil, a collection of strongly regular graphs that had been used by D. Corneil and others. The first seven of these graphs each had 25 points. Using the faster version of the refinement procedure, our algorithm took 1964, 1392, and 1652 sec respectively on the first three of these graphs On the fourth and fifth it failed to find all isomorphisms within 3000 sec, and we did not consider it worthwhile to run the algorithm with further graphs in this collection, since the first five took so long. The adjacency matrix for the first graph is

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

In the case of graph (not subgraph) isomorphism, in the refinement procedure we could use, as well as (2), the inverse condition $m_{ij} = 1$ only if

$$\underset{1\le y\le p_\beta}{(\forall y)} \quad (b_{yj} = 1) \Rightarrow \quad \underset{1\le x\le p_\alpha}{(\exists x)} \quad (a_{ix}\cdot m_{xy} = 1).$$

However, this condition is not mathematically indispensible and we did not use it because it would not have allowed exploitation of the parallelism that was mentioned in Section 3.

Perhaps we should also mention that in the case of graph isomorphism, if a matrix M' is unchanged by the refinement procedure, then G_A is isomorphic to G_B. To see this, we can reason as in Section 3 to establish that if refinement leaves M' unchanged, then if two points are adjacent in G_A the corresponding points in G_B are adjacent. When M^0

is constructed according to (3) the degrees of corresponding points are the same, so there can be no two adjacent points in G_B that correspond to nonadjacent points in G_A, if M' is unchanged by the refinement procedure.

DIGRAPH ISOMORPHISM. Berztiss [2] has provided a graph and subgraph isomorphism procedure for directed graphs. A *directed graph* or *digraph* consists of a finite nonempty set V of points, together with a set of ordered pairs of points in V. The algorithm introduced in the present paper is similar to that of Berztiss in that it works by tree searching instead of vertex classification, and it is an algorithm, not merely a heuristic procedure, and not based on an unproven conjecture. Berztiss represents digraphs by linear formulas, and his algorithm constructs successively larger subformulas that match in the two digraphs. Our algorithm differs in that at any stage of the search, we are not concerned only with a subset of the rows of M: every 1 in M is processed every time the refinement procedure is executed. In Berztiss' algorithm there is no obvious counterpart of the iteration of the refinement procedure. Since our algorithm works directly with adjacency matrices, we do not have to construct linear formulas.

To compare the timing of our procedure with that of Berztiss, we have experimented with the same family of digraphs that Berztiss used. These are digraphs in which the indegree and outdegree of every point is exactly $p_\alpha/2$. To test for digraph isomorphism, we start by constructing M^0 according to

$$m^0_{i,j} = \begin{cases} 1 \text{ if the indegree of the } i\text{th point in } G_A \text{ is the same as the indegree of the } j\text{th} \\ \quad \text{point in } G_B \text{ and the outdegree of the } i\text{th point in } G_A \text{ is the same as the out-} \\ \quad \text{degree of the } j\text{th point in } G_B, \\ 0 \text{ otherwise.} \end{cases}$$

We then apply the algorithm of Section 3 with the refinement procedure modified as follows: $m_{i,j} = 1$ is changed to $m_{i,j} = 0$ unless

$$(\underset{1\le x\le p_\alpha}{\forall x})\,((a_{ix} = 1) \Rightarrow (\underset{1\le y\le p_\beta}{\exists y})\,(m_{xy}\, b_{jy} = 1)) \quad \text{and} \quad (\underset{1\le x\le p_\alpha}{\forall x})\,((a_{xi} = 1) \Rightarrow (\underset{1\le y\le p_\beta}{\exists y})\,(m_{xy}\cdot b_{yj} = 1)).$$

For nonisomorphic and isomorphic digraphs our experimental results are summarized in Table III. There appears to be no significant difference between the timings on isomorphic and nonisomorphic digraphs, and the timing increases with p_α less rapidly than that of Berztiss' algorithm.

5. *Parallel Hardware Embodiment of Refinement Procedure*

We now describe a logic-in-memory array that can execute the refinement procedure in less than one μsec. This hardware is remarkably simple, but it requires a very large number of gates.

It is convenient to regard A, B, and M as Boolean matrices in which 1 and 0 correspond to *true* and *false* respectively. Further, it is convenient to define a $p_\alpha \times p_\beta$ Boolean matrix $R = [r_{xj}]$ by

$$r_{xj} = (\exists_y)(m_{xy}\cdot b_{yj}). \tag{4}$$

TABLE III. RESULTS OF EXPERIMENTS WITH DIGRAPH ISOMORPHISM

p_α	Time in seconds			
	Nonisomorphic		Isomorphic	
	av.	s d.	av	s d
6	0 3	0 5	0.3	0 5
8	0.8	0.6	1 0	0 9
10	2.5	1 1	2.8	1.2
12	6 5	2 4	6.4	2.4

Condition (2) can now be written as $m_{ij} = m_{ij} \cdot (\forall x)(\bar{a}_{ix} \vee r_{xj})$, which can readily be manipulated into

$$m_{ij} = m_{ij} \cdot \overline{(\exists x)(a_{ix} \cdot \bar{r}_{xj})}. \tag{5}$$

The hardware includes a separate bistable element (flip-flop) corresponding to each element of each of the matrices A, B, and M. For instance, as in [6], the bistable corresponding to a_{ij} is set to state 1 if $a_{ij} = 1$ or to state 0 if $a_{ij} = 0$, we shall not discuss the means by which this is done. For each $x = 1, \cdots, p_\alpha$ and $y, j = 1, \cdots, p_\beta$ there is a separate *and* gate that derives its two inputs from the bistables that correspond to m_{xy} and b_{yj}. The network also includes one *or* gate for each $x = 1, \cdots, p_\alpha$ and $j = 1, \cdots, p_\beta$. Any such *or* gate computes $r_{xj} = (\exists x)(m_{xy} \cdot b_{yj})$ and its p_β inputs are derived from the outputs of the *and* gates that compute $m_{xy} \cdot b_{yj}$ for $y = 1, \cdots, p_\beta$. Corresponding to each *or* gate r_{xj} there is an inverter whose output is $\bar{r}_{xj}$. For each x, $i = 1, \cdots, p_\alpha$ and $j = 1, \cdots, p_\beta$ there is an *and* gate that computes $a_{ix} \cdot \bar{r}_{xj}$. Finally, the network includes $p_\alpha \times p_\beta$ *or* gates, each of which computes $(\exists x)(a_{ix} \cdot \bar{r}_{xj})$ for different i, j. The inputs to the *or* gate that computes $(\exists x)(a_{ix} \cdot \bar{r}_{xj})$ are the outputs of the p_α *and* gates that compute $a_{ix} \cdot \bar{r}_{xj}$ for $x = 1, \cdots, p_\alpha$. Perhaps because the network is essentially four-dimensional (i, j, x, y), we have not been able to produce a really helpful diagram of it.

The network operates as follows. At time t_0 the matrix M is read into the $p_\alpha \times p_\beta$ bistables m_{ij} that are provided for it. At a time t_1 that is sufficiently delayed after t_0 to allow operation of all of the *or* gates r_{xj}, the external inputs to bistables m_{ij} are removed. Thereafter the bistable m_{ij} is reset to state 0 if the *or* gate $(\exists x)(a_{ix} \cdot \bar{r}_{xj})$ produces output 1, and otherwise the state of the bistable m_{ij} remains unchanged. At a later time t_2 the matrix M that results from the refinement procedure can be read out from the bistables m_{ij}. The refinement procedure is executed asynchronously, and time t_2 must be sufficiently delayed after t_1 to allow completion of the asynchronous iterative computation.

6. *Boolean Matrix Formulation of the Refinement Procedure*

For the purposes of the present section we regard (4) as the definition of a Boolean product $R = [r_{xj}] = M \times B$, and we also use the notation $\bar{M} = [\bar{m}_{ij}]$ and $M \cdot M' = [m_{ij}\ m'_{ij}]$. Using (5), this allows us to formulate the following refinement procedure:

Step 1 $R = M \times B$,
Step 2 $M = M\ \overline{(A \times \bar{R})}$,
Step 3. If any row of M contains no 1's then go to FAIL exit;
Step 4. If M was changed by step 2 then go to step 1 else go to SUCCEED exit,

We have introduced this formulation because it is succinct, but unfortunately it does not express the asynchronous nature of the refinement procedure. This formulation suggests that step 2 is carried out only after completion of step 1. In our software and hardware implementations, m_{ij} is changed to $m_{ij} = 0$ as soon as condition (2) is not satisfied: there would be no practical advantage in postponing such changes until the end of an iteration.

7. *Conclusion*

For isomorphic random graphs our algorithm finds all isomorphisms in a time roughly proportional to p_α^3, and this satisfies Corneil and Gotlieb's criterion that an algorithm is efficient if the time is proportional to a power of p_α. However, for the very limited classes of graphs that we have used experimentally, our graph isomorphism algorithm appears to be less efficient than that of Corneil and Gotlieb [4] and our clique detection procedure is probably less efficient than that of Bron and Kerbosch [3]. The principal advantage of our algorithm is that it can cope with undirected subgraph isomorphism, although this may be a slow process when p_α and p_β are large For instance, we abandoned subgraph isomorphism experiments with $p_\alpha = 10$ and $p_\beta = 15$ because fifty trials would have been costly; and substantially less than fifty trials would not have given a worth-

while estimate of the average time per trial. The slowness of the algorithm in this case can be partly attributed to the large values of q_β that result from *or*'ing adjacency matrix A into adjacency matrix B. The algorithm obviously works more quickly the sparser the matrix M^0, and this is why the algorithm is more efficient for graph isomorphism and clique detection than for general subgraph isomorphism.

It is perhaps unnecessary to discuss the obvious elaborations of the algorithm for application to n-ary relational structures, where each such structure consists of a set of V points together with a set of n-tuples of points in V.

Appendix 1. Statement of the Refinement Procedure

In the following formulation, h, i, j, k, and x are integer pointers. elim is an integer whose value is the number of 1's that have been eliminated so far in the present iteration. $\deg_i$ is the degree of the ith point in G_A. lst is a list of all points that are adjacent to the ith point in G_A. sc is a word that contains only one 1, which is used for scanning. A_i contains the ith row of matrix A, B_j contains the jth column of matrix B, and M_i contains the ith row of matrix M. In each case the rightmost or bottommost bit in a matrix row or column is located at the least significant bit of the computer word that contains it. & means collation, e.g. 1100 & 1010 = 1000. NOT means negation of all bits, e.g. NOT 1100 = 0011. We have programmed the refinement procedure as follows:

Step 1 elim := 0,
$i := 1$;
Step 2 $k := 1$,
sc := $2^{(p_\alpha - 1)}$;
$h := 1$;
Step 3. if sc & $A_i = 0$ then go to step 4;
$\text{lst}_k = h$,
$k := k + 1$;
Step 4. sc := sc $\times 2^{-1}$,
$h := h + 1$;
if $k \neq \deg_i + 1$ then go to step 3;
Step 5. $j = 1$,
sc = $2^{(p_\beta - 1)}$,
Step 6. if M_i & sc = 0 then go to step 9;
$h := 1$,
Step 7. $x := \text{lst}_h$,
if M_x & $B_j = 0$ then go to step 8;
$h = h + 1$;
if $h \neq \deg_i + 1$ then go to step 7 else go to step 9;
Step 8. $M_i = M_i$ & NOT sc,
elim := elim + 1;
$h := h + 1$;
Step 9. sc := sc $\times 2^{-1}$;
$j = j + 1$;
if $j \neq p_\beta + 1$ then go to step 6;
Step 10 if $M_i = 0$ then go to FAIL exit;
$i = i + 1$,
if $i \neq p_\alpha + 1$ then go to step 2;
if elim $\neq 0$ then go to step 1;
go to SUCCEED exit;

At the expense of using more storage area, the array lst can of course be set up once-and-for-all at the start of the isomorphism program, so that it is unnecessary to repeat steps 2, 3, and 4 in each iteration of the refinement procedure; computer time is thereby saved.

Appendix 2. Modification of the Isomorphism Algorithm for Use in Clique Detection

For use in clique detection we modified the algorithm of Section 3 by changing "if $d = 1$ then $k := H_1$ else $k := 0$;" to "if $d = 1$ then $k := H_1$ else $k := H_{d-1}$;" in step 2.

A consequence of this modification is that in any generated matrix M', $j_i < j_{i+1}$ where j_i is the value of j such that $m'_{ij} = 1$. Dr. B. R. Heap pointed out that this fact can be used to speed up the program by removing 1's such that $j_i \geq j_{i+1}$ from matrices M. In all our experiments with clique detection, whenever the refinement procedure was executed, it was immediately preceded by

```
if d < p then
begin c := k; for c := d + 1 step 1 until p_α do
  begin for f := 1 step 1 until c do m_ef := 0;
    c := c + 1;
  end
end;
```

When this was used in step 1 of the algorithm of Section 3, we set $k := 1$.

If the algorithm of Section 3 had been used without modification, then when a clique was found the algorithm would also have found all of the isomorphisms between G_α and this clique, thus in effect enumerating the automorphisms of the clique. We were not interested in the number of automorphisms of a clique; we only wanted to know the number of distinct cliques in G_β. The modification works by ensuring that in any generated matrix M', $j_1 < j_2 < \cdots < j_{p_\alpha}$ where $j_1, j_2, \cdots, j_{p_\alpha}$ are the values of j corresponding respectively to the p_α 1's in M': the modification precludes all other permutations.

ACKNOWLEDGMENTS. The experiments with cliques were carried out at the suggestion of Dr. B. R. Heap, who also proposed the modifications described in Appendix 2. The author is grateful to the referee for his comments and, in particular, for recommending comparison with References [2] and [3].

REFERENCES

1. BARROW, H. G., AMBLER, A. P., AND BURSTALL, R. M. Some techniques for recognising structures in pictures. In *Frontiers of Pattern Recognition*, S. Watanabe, Ed., Academic Press, New York, 1972, pp. 1–29
2. BERZTISS, A. T. A backtrack procedure for isomorphism of directed graphs. *J. ACM 20*, 3 (July 1973), 365–377.
3. BRON, C., AND KERBOSCH, J. Algorithm 457. Finding all cliques in an undirected graph [H] *Comm. ACM 16*, 9 (Sept. 1973), 575–577
4. CORNEIL, D. G., AND GOTLIEB, C. C. An efficient algorithm for graph isomorphism *J. ACM 17*, 1 (Jan. 1970), 51–64.
5. HARARY, F. *Graph Theory*. Addison-Wesley, Reading, Mass., 1969.
6. LEVITT, K. N., AND KAUTZ, W. H. Cellular arrays for the solution of graph problems *Comm. ACM 15*, 9 (Sept. 1972), 789–801
7. NILSSON, N. J. *Problem-Solving Methods in Artificial Intelligence*. McGraw-Hill, New York, 1971
8. OSTEEN, R. E., AND TOU, J. T. A clique-detection algorithm based on neighbourhoods in graphs. *Int. J. of Comput. and Inform. Scis. 2*, 4 (Dec. 1973), 257–268.
9. PIKE, M. C., AND HILL, I. D. Algorithm 266: Pseudo-random numbers [G5]. *Comm. ACM 8*, 10 (Oct. 1965), 605–606
10. SAKAI, T., NAGAO, M., AND MATSUSHIMA, H. Extraction of invariant picture substructures by computer. *Comput. Graphics and Image Process. 1*, 1 (April 1972), 81–96.
11. ULLMANN, J. R. *Pattern Recognition Techniques*. Butterworths, London, and Crane Russak, New York, 1973.

RECEIVED MARCH 1974; REVISED FEBRUARY 1975