International Journal of Computer and Information Sciences, Vol. 5, No. 3, 1976

Cliques of a Graph—Variations on the Bron–Kerbosch Algorithm

H. C. Johnston[1]

Received March 1975; revised December 1975

This paper develops a family of algorithms that are variations on the Bron–Kerbosch algorithm for finding all the cliques of a simple undirected graph. The algorithms are developed in a stepwise manner, from a recursive algorithm for generating all combinations of zero or more objects chosen from N objects. Experimental results are given.

KEY WORDS: Cliques; combinatorial programming; graph theory; stepwise refinement; tree search.

1. INTRODUCTION

A clique is a maximal, completely connected subgraph of a simple undirected graph. Some authors call *any* completely connected set of vertices a clique, but we use the term only for *maximal* completely connected sets, i.e., those having no completely connected supersets. We arbitrarily assume that no vertex of the graph is connected to itself.

This paper develops, by a form of stepwise program development, a family of programs for generating all the cliques of a graph. One member of the family is equivalent to the algorithm published by Bron and Kerbosch.[4]

After the algorithms have been obtained in a concise, abstract form, modifications are considered that increase the efficiency by reducing the

This work was supported in part by National Science Foundation grant DCR 72-03752 AO2.

[1] Department of Computer Science, Queen's University of Belfast, Belfast, Northern Ireland.

number of procedure calls needed. Experimental results from concrete implementations are presented and conclusions drawn from them.

The paper should be of value to the following two groups of readers:

1. those with an interest in algorithm design and presentation, particularly in the field of combinatorial programming;
2. those with a particular interest in the efficient generation of cliques.

2. ABSTRACT PROGRAMMING LANGUAGE

To help develop the algorithms, an abstract programming language is required that allows us to have sets and graphs as basic data structures and provides appropriate methods for operating on them. To achieve this we use an ALGOL-like language with the following features:

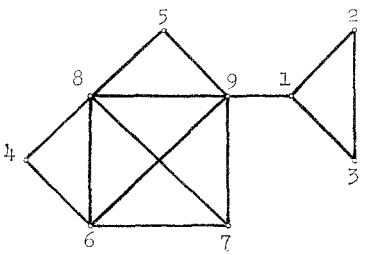
1. variables of type **set**, where for simplicity we assume that the members of a set are positive integers;
2. the set operators $\cap$ (intersection), $\cup$ (union), $\neg$ (complement), $=$ (equality), and $\neq$ (inequality);
3. $\{\ \}$ to denote the empty set;
4. $\{i\}$ to denote the unit set with member i;
5. $\{i \cdots j\}$ to denote the set with members $i, i+1, i+2, \ldots, j-1, j$;
6. ANY MEMBER IN (s) to denote an arbitrary member in the set s (undefined if s is empty);
7. SIZE OF (s) to denote the number of members in the set s;
8. variables of type **graph**, where by a graph we mean a simple undirected graph whose vertices are denoted by positive, not necessarily consecutive, integers;
9. VERTICES OF (g) to denote the set of all vertices of the graph g;
10. $g[i]$ to denote the set of vertices of the graph g that are connected to vertex i by an edge.

In the above, the definition of the operator ANY MEMBER IN is nondeterministic since it does not state how the selected member is to be chosen from s. This reflects the fact that the members of a set do not have a natural order; thus, for example, the phrase "the first member of a set" would be meaningless. By choosing to denote the members of sets by positive integers, we have imposed a natural order on them, but the correctness of the algorithms that we develop does not depend in any way on this imposed order.

In connection with point 10 above, we note that a set is the appropriate data type for the vertices connected to a given vertex only because we are dealing with simple graphs. If our graphs were allowed to have parallel edges, a more complex data structure would be needed for $g[i]$.

3. REPRESENTATION OF THE GRAPH

It is important that we do *not* make any assumptions about the methods to be used within the computer for representing a graph, or any other data structure, until we have established the algorithms that we wish to implement.



(a)

	1	2	3	4	5	6	7	8	9
1	0	1	1	0	0	0	0	0	1
2	1	0	1	0	0	0	0	0	0
3	1	1	0	0	0	0	0	0	0
4	0	0	0	0	0	1	0	1	0
5	0	0	0	0	0	0	0	1	1
6	0	0	0	1	0	0	1	1	1
7	0	0	0	0	0	1	0	1	1
8	0	0	0	1	1	1	1	0	1
9	1	0	0	0	1	1	1	1	0

(b)

G[1] = {2,3,9}
G[2] = {1,3}
G[3] = {1,2}
G[4] = {6,8}
G[5] = {8,9}
G[6] = {4,7,8,9}
G[7] = {6,8,9}
G[8] = {4,5,6,7,9}
G[9] = {1,5,6,7,8}

(c)

Fig. 1

We will then know which operations we need to perform on the various data items and will be able to select the concrete representations wisely.

However, we require an example graph to illustrate various stages of our discussion; thus, in Fig. 1, we display a graph with nine vertices represented in various ways. The same example was used by Osteen and Tou.[12] Part (a) of the figure shows the conventional diagrammatic representation of the graph. Part (b) shows the *connection matrix* or adjacency matrix of the graph. For a graph with N vertices this is a square matrix G with N rows and columns which has $G[I, J] = 1$ if the graph contains an edge connecting vertex I to vertex J and $G[I, J] = 0$ otherwise. Part (c) represents the graph by a vector $G[1{:}9]$ with each element a subset of the integers $1 \cdots 9$. In this case, $I \in G[J]$ if the graph contains an edge connecting vertex I to vertex J and $I \notin G[J]$ otherwise. Of these three methods, the latter relates most directly to the notation that we have defined in our abstract programming language.

We note that for this particular problem the following properties are satisfied:

1. $I \in G[J] \subset J \in G[I]$ (1)

since we are dealing with undirected graphs.

2. $I \notin G[I]$ (2)

since we have assumed that no vertex is connected to itself.

4. EVOLUTION OF THE ALGORITHMS

In this section we start with two simple algorithms—one for finding a single clique and the second for finding all combinations of zero or more objects chosen from a given set of objects—and, in a stepwise manner, evolve toward the algorithms for finding all the cliques of a graph. A similar process of evolution can be applied to other combinatorial problems.

4.1. Find and Use One Clique (Table I)

As a first step, consider how to find a single clique. Let I and J be any two vertices of a graph G and consider the subset of vertices obtained by taking the intersection of $G[I]$ with $G[J]$. Clearly this subset defines those vertices that are connected to both vertex I and vertex J. For example, taking $I = 6$ and $J = 8$ in the graph defined in Fig. 1 we get

$$G[6] \cap G[8] = \{4, 7, 8, 9\} \cap \{4, 5, 6, 7, 9\} = \{4, 7, 9\}$$

Table I. An Algorithm to Find One Clique

```
A1    procedure FIND AND USE ONE CLIQUE (value graph G; procedure USE);
A2    begin set CLIQUE, INTERSECTION;
A3      CLIQUE := { }; INTERSECTION := VERTICES OF (G);
A4      repeat integer I;
A5        I := ANY MEMBER IN (INTERSECTION);
A6        CLIQUE := CLIQUE ∪ {I};
A7        INTERSECTION := INTERSECTION ∩ G[I];
A8      until INTERSECTION = { };
A9      USE(CLIQUE);
A10   end FIND AND USE ONE CLIQUE;
```

which shows that vertices 4, 7, and 9 are the only ones connected to both vertex 6 and vertex 8. We may use this to build up a single clique by using the simple algorithm expressed by the procedure FIND AND USE ONE CLIQUE in Table I. The parameter G defines the graph and USE is a procedure that will output, store, or otherwise use the clique as a particular application requires.

The validity of this algorithm depends on the following facts:

1. Each time the **repeat** loop is entered the members of the set INTERSECTION are those vertices that are connected to every vertex that is a member of CLIQUE. This is certainly true after the initialization statements A3, and its truth on subsequent loops is ensured by statement A7 each time the member I is added to the clique.

2. Each new member put into CLIQUE is known to be connected to every member in CLIQUE since statement A5 selects the new member from INTERSECTION.

3. The loop must terminate, since each execution of it must remove at least one member from INTERSECTION, namely the member I since $I \notin G[I]$; thus it can be executed a maximum of N times.

4. On exit from the **repeat** loop, INTERSECTION $= \{ \}$. Thus there is no vertex connected to all the vertices already in CLIQUE. This ensures that the value of CLIQUE passed to the procedure USE is a maximal completely connected set of vertices. At earlier stages CLIQUE is a completely connected set of vertices, but not yet maximal.

Applying this algorithm to our example graph and assuming that the procedure ANY MEMBER IN selects the numerically smallest member of its argument, we get the sequence of values for I, CLIQUE, and INTERSECTION shown in Table II.

Table II. Application of the Algorithm in Table I to the Graph in Fig. 1

I	CLIQUE	INTERSECTION
—	{ }	{1, 2, 3, 4, 5, 6, 7, 8, 9}
1	{1}	{2, 3, 9}
2	{1, 2}	{3}
3	{1, 2, 3}	{ }

4.2. Find and Use All Combinations (Table III)

In order to develop an algorithm for finding all the cliques of a graph, let us consider an algorithm for finding all possible combinations of zero or more objects chosen from a given set of objects V. This is a reasonable starting point since each clique is a combination chosen from the vertices of the graph. An appropriate algorithm is shown in Table III. It is expressed as

Table III. An Algorithm to Find All Combinations[a]

		Analysis
B1	**procedure** FIND AND USE ALL COMBINATIONS (**value set** V; **procedure** USE);	1
B2	**begin**	1
B3	**procedure** EXPAND (**value set** COMBINATION, POSSIBLE);	n
B4	**begin set** S;	n
B5	USE (COMBINATION);	n
B6	$S :=$ POSSIBLE;	n
B7	**while** $S \neq \{\ \}$ **do**	$2n - 1$
B8	**begin integer** I;	$n - 1$
B9	$I :=$ ANY MEMBER IN(S); $S := S \cap \neg \{I\}$;	$n - 1$
B10	POSSIBLE := POSSIBLE $\cap \neg \{I\}$;	$n - 1$
B11	EXPAND (COMBINATION $\cup \{I\}$, POSSIBLE);	$n - 1$
B12	**end**;	$n - 1$
B13	**end** EXPAND;	n
B14	EXPAND ({ }, V);	1
B15	**end** FIND AND USE ALL COMBINATIONS;	1

[a] Note: $n = 2^N$, where N is the size of the set V.

a procedure FIND AND USE ALL COMBINATIONS with the set V as a parameter and with a parameter USE as in FIND AND USE ONE CLIQUE.

This algorithm performs a depth-first tree search. Its fundamental structure is very similar to that of many other tree-searching algorithms (see, for example, Burstall[6]). The desired result is achieved by a single call to a recursive procedure EXPAND which has two parameters:

COMBINATION:	a set of integers that records the combination currently under consideration; initially COMBINATION = { };
POSSIBLE:	a set of integers that records the objects which remain as possible future members of COMBINATION; initially POSSIBLE = V.

Each call of EXPAND first outputs the current combination and then, by calling itself recursively, finds all combinations of members of V that contain COMBINATION and are contained in COMBINATION ∪ POSSIBLE.

In this algorithm the local variable S is not essential. Its value is always the same as the value of POSSIBLE; thus each occurrence of S could be replaced by POSSIBLE. The variable S has been introduced to simplify the evolution of the algorithms that follow.

If the total number of elements in the set V is N and we let n denote the number of nodes on the search tree, which is exactly equal to the number of calls of EXPAND and to the number of combinations found, then we have

$$n = 2^N \tag{3}$$

An analysis of the number of times each line of the algorithm is executed, expressed in terms of n, is given in Table III.

If the procedure ANY MEMBER IN delivers the numerically smallest member of its argument, the combinations are generated in lexicographical order. For example, with $V = \{1, 2, 3\}$ the sequence produced is

$$\{\ \}, \{1\}, \{1, 2\}, \{1, 2, 3\}, \{1, 3\}, \{2\}, \{2, 3\}, \{3\}$$

4.3. Find and Use All Cliques (Table IV)

What must we do to FIND AND USE ALL COMBINATIONS so that it produces cliques instead? First, we take V to be the set of vertices of the graph, or a subset of them if we only want the cliques of a subgraph. Then, if there were an easy test that could be applied to a given combination to see whether it is a clique, we could simply make the use of a given com-

bination dependent on the result of this test. For example, we could replace statement B5 by

if COMBINATION IS A CLIQUE **then** USE (COMBINATION); (4)

where COMBINATION IS A CLIQUE is a Boolean procedure that delivers the value **true** when COMBINATION is a clique and the value **false** otherwise.

However, we can do much better than this. Looking back at FIND AND USE ONE CLIQUE we see that, by putting a vertex which is a member of INTERSECTION into CLIQUE at each stage, we ensure that at all stages the set generated represents a completely connected set of vertices. We can apply the same constraint to the sets generated by FIND AND USE ALL COMBINATIONS by removing from POSSIBLE all vertices not connected to vertex I when we put vertex I into COMBINATION. Thus a vertex chosen to be put into COMBINATION will always be connected to each of the existing members. To make this change we introduce the graph G as a parameter of the main procedure and replace statement B11 by

$$\text{EXPAND(COMBINATION} \cup \{I\}, \text{POSSIBLE} \cap G[I])$$

All that now remains for the test COMBINATION IS A CLIQUE in (4) to do is to ensure the maximality property required of a clique. Again, from FIND AND USE ONE CLIQUE, we know that the set is maximal when INTERSECTION $= \{\ \}$. Thus, if we can compute

$$\text{INTERSECTION} = \bigcap_{I \in \text{COMBINATION}} G[I]$$

then in place of (4) we can simply have

if INTERSECTION $= \{\ \}$ **then** USE(COMBINATION);

We *cannot* use POSSIBLE in place of INTERSECTION in this test since POSSIBLE is in general a subset of INTERSECTION due to the members removed from it by statement B10. The most efficient way to evaluate INTERSECTION is to build it up with COMBINATION. To do this we introduce INTERSECTION as a redundant parameter in the procedure EXPAND, with initial value V, and update it each time we put a new member into COMBINATION.

Since INTERSECTION $= \{\ \}$ implies that POSSIBLE $= S = \{\ \}$, it is clear that the procedure EXPAND either presents a clique for use *or* expands the existing incomplete clique. This can be made more explicit, and at the same time the effort of calculating S when INTERSECTION $= \{\ \}$ can be saved, by introducing an **if then else** construction.

Table IV. An Algorithm to Find All Cliques[a]

		Analysis
C1	**procedure** FIND AND USE ALL CLIQUES (**value set** V; **value graph** G; **procedure** USE);	1
C2	**begin**	1
C3	**procedure** EXPAND (**value set** COMBINATION, INTERSECTION, POSSIBLE);	$n + c$
C4	**if** INTERSECTION $= \{\ \}$ **then** USE(COMBINATION) **else**	$n + c : c$
C5	**begin set** S;	n
C6	CALCULATE S;	n
C7	**while** $S \neq \{\ \}$ **do**	$2n + c - 1$
C8	**begin integer** I;	$n + c - 1$
C9	$I :=$ ANY MEMBER IN (S); $S := S \cap \neg \{I\}$;	$n + c - 1$
C10	POSSIBLE := POSSIBLE $\cap \neg \{I\}$;	$n + c - 1$
C11	EXPAND (COMBINATION $\cup \{I\}$, INTERSECTION $\cap\ G[I]$, POSSIBLE $\cap\ G[I]$);	$n + c - 1$
C12	**end**;	$n + c - 1$
C13	**end** EXPAND;	n
C14	EXPAND($\{\ \}$, V, V);	1
C15	**end** FIND AND USE ALL CLIQUES;	1

[a] Note: 1. n and c are defined in (5) and (6).
2. The notation used in the analysis column of line C4 means that the first part of the line is executed $n + c$ times and the second part just c times.

Making these changes to Table III, we evolve the procedure FIND AND USE ALL CLIQUES in Table IV. We have also replaced statement B6 by

CALCULATE S;

where the procedure CALCULATE S is given in Table V(a). We refer to this method of calculating S as *the basic method.* Further methods are discussed below.

We can analyze the number of times each line of this algorithm is executed in terms of the quantities

c: the number of cliques found; (5)

and

n: the number of nodes in the search tree *excluding* the c nodes at which cliques are found. (6)

Such an analysis is given in Table IV.

Table V. Several Methods for Calculating the Set S

```
(a)  D1   procedure CALCULATE S; comment The Basic Method;
     D2     S := POSSIBLE;
(b)  E1   procedure CALCULATE S;
     E2   comment The Generalized Bron–Kerbosch Method;
     E3   begin integer THRESHOLD, SIZE OF S; set T;
     E4     THRESHOLD := ?;
     E5     comment the ? may be replaced by any integer ⩾ 0;
     E6     SIZE OF S := ∞;
     E7     T := INTERSECTION;
     E8     repeat integer J, NEW SIZE; set NEW S;
     E9       J := ANY MEMBER IN(T); T := T ∩ ¬ {J};
     E10      NEW S := POSSIBLE ∩ ¬ G[J];
     E11      NEW SIZE := SIZE OF (NEW S);
     E12      if NEW SIZE < SIZE OF S then
     E13      begin S := NEW S; SIZE OF S := NEW SIZE end;
     E14    until T = { } or SIZE OF S ⩽ THRESHOLD;
     E15  end CALCULATE S;
(c)  F1   procedure CALCULATE S;
     F2   comment The Simplified Bron–Kerbosch Method;
     F3     S := POSSIBLE ∩ ¬ G[ANY MEMBER IN (INTERSECTION)];
```

Since our modifications have done nothing to alter the order in which the combinations are generated, this algorithm generates the cliques in lexicographical order *provided* that the procedure ANY MEMBER IN delivers the numerically lowest member of its argument.

4.4. Reduction of the Search Tree

The next stage of evolution is to see whether we can reduce the size of the tree searched while retaining the completeness of the search. The question that we must ask is: Can we find all of the solutions without searching all of the tree? We show that the answer to this question is "yes."

First, we note that the correctness of the algorithm is independent of the order in which the **while** loop C7-12 selects and processes the members of the set S. We are, therefore, free to choose a particular partial ordering as follows:

1. Select a vertex J such that

$$J \in \text{INTERSECTION} \tag{7}$$

2. Partition the set S into two disjoint sets defined by

$$S_1 = S \cap \neg\, G[J] \tag{8}$$

$$S_2 = S \cap \neg\, S_1 = S \cap G[J] \tag{9}$$

Then the vertices in S_1 are *not* connected to vertex J while those in S_2 are so connected.

3. Take all the members of S_1 before the members of S_2.

Now suppose that we interrupt the processing of the set S at the point at which all the members of S_1 have been processed but no member of S_2 has yet been processed. Let P_0 be the value of POSSIBLE on entry to the current call of EXPAND and let P_1 be the value at our point of interruption. Then, since each vertex is removed from POSSIBLE by statement C10 as it is processed, we have

$$P_1 = P_0 \cap \neg\, S_1 = S \cap \neg\, S_1 = S_2$$

It follows that, if we now resume the processing, every vertex I chosen for inclusion in COMBINATION by the remaining calls of EXPAND at the current level, and in all enclosed calls, must satisfy

$$I \in S_2$$

But, by definition (9),

$$S_2 \subset G[J]$$

Thus

$$I \in G[J]$$

which, using (1), gives

$$J \in G[I]$$

and combining this with (7) we get

$$J \in \text{INTERSECTION} \cap G[I].$$

This means that, no matter how many members in S_2 are put into COMBINATION, the corresponding value of INTERSECTION will always retain the member J. Thus, throughout the section of our search after the interruption, INTERSECTION cannot become empty and, therefore, no new cliques will be found. Since our resumed search would therefore be rather fruitless, we can simply remove the members of S_2 from S at the outset. The required change is effected by calculating S from an expression of the form

$$S := \text{POSSIBLE} \cap \neg\, G[J] \tag{10}$$

where J is *any* member in INTERSECTION.

It is important to note that this change removes certain vertices from S but does *not* remove them from POSSIBLE. Thus the variables S and POSSIBLE no longer have the same values. A vertex that is a member of S_2 remains in POSSIBLE; thus it can become a member of COMBINATION in a call of EXPAND enclosed by the current call.

This change reduces the size of the set S in some or all of the calls of EXPAND and thus reduces the size of the search tree. Because it also disturbs the order in which vertices of the graph are allowed to enter the set COMBINATION, the cliques are no longer generated in lexicographical order. Normally this is not a disadvantage.

4.5. Introduction of THRESHOLD [Table V(b), (c)]

It remains to select the special vertex J from the set INTERSECTION. A range of methods, distinguished by the value of a variable called THRESHOLD, are considered. All of them are embodied in the version of CALCULATE S shown in Table V(b). In this procedure, the following points are of interest:

1. The body of the **repeat** loop, statements E9-13, selects a member J in INTERSECTION (initially $T =$ INTERSECTION) and calculates the set corresponding to (10) for it. For the first values of J, this set becomes the initial value of S. Thereafter, the value of S is changed only if a set of smaller size is found.

2. The number of times the **repeat** loop is executed depends on the value given to the variable THRESHOLD. If we set THRESHOLD $= \infty$ then the condition SIZE OF $S \leqslant$ THRESHOLD will be **true** after the first value of J, so no further values will be examined. If we set THRESHOLD $= 0$ then looping will continue either until $T = \{\ \}$, i.e., all the members of INTERSECTION have been tried, or until SIZE OF $S = 0$ i.e., $S = \{\ \}$. If we set THRESHOLD $= 4$, say, then values of J will be examined until $T = \{\ \}$ or SIZE OF $S \leqslant 4$, whichever occurs first.

3. The procedure is valid no matter what value of THRESHOLD we use because the **repeat** loop is always executed at least once and we know that the set S calculated from *any* value of J will do, as far as the rest of the algorithm is concerned.

Readers familiar with other clique-finding algorithms will have recognized that when THRESHOLD $= 0$ the algorithm we have evolved is almost equivalent to the Bron–Kerbosch algorithm.[4] (The author is indebted to H. G. Barrow for pointing out this fact.) We will therefore refer to this family of algorithms as the *generalized Bron–Kerbosch algorithms*.

If THRESHOLD $= \infty$, the procedure CALCULATE S can be simplified to the version shown in Table V(c). We will refer to this as the *simplified Bron–Kerbosch algorithm*. This is *not* equivalent to the earlier algorithm published by Bron, Kerbosch, and Schell.[5]

A comparison of Tables III and IV with the published version of the Bron–Kerbosch algorithm is made easier by noting that

1. They arbitrarily assume that each vertex of the graph *is* connected to itself; thus their Boolean matrix CONNECTED has the diagonal elements **true**: the current algorithms assume that each vertex *is not* connected to itself.

2. The following relationship exists between the sets CANDIDATES and NOT used by them and the sets INTERSECTION and POSSIBLE in the current algorithms:

$$\text{CANDIDATES} = \text{POSSIBLE}$$
$$\text{NOT} = \text{INTERSECTION} \cap \neg \text{POSSIBLE}$$

Table VI. Application of the Generalized Bron–Kerbosch Method with THRESHOLD = 0 to the Graph in Fig. 1

```
EXPAND({ }, {123456789}, {123456789});
J = 8; S = {1238};
        I = 1; EXPAND({1}, {239}, {239});
                J = 2; S = {29};
                        I = 2; EXPAND({12}, {3}, {3});
                                J = 3; S = {3};
                                        I = 3; EXPAND({123}, { }, { });
                                                USE({123});
                        I = 9; EXPAND({19}, { }, { });
                                USE({19});
        I = 2; EXPAND({2}, {13}, {3});
                J = 1; S = { };
        I = 3; EXPAND({3}, {12}, { });
                J = 1; S = { };
        I = 8; EXPAND({8}, {45679}, {45679});
                J = 6; S = {56};
                        I = 5; EXPAND({58}, {9}, {9});
                                J = 9; S = {9};
                                        I = 9; EXPAND({589}, { }, { });
                                                USE({589});
                        I = 6; EXPAND({68}, {479}, {479});
                                J = 4; S = {47};
                                        I = 4; EXPAND({468}, { }, { });
                                                USE({468});
                                        I = 7; EXPAND({678}, {9}, {9});
                                                J = 9; S = {9};
                                                        I = 9; EXPAND({6789}, { }, { });
                                                                USE({6789});
```

3. They have chosen to represent the various sets used in the algorithm by integer arrays: no assumptions about representation have so far been made in the current algorithms.

The application of the generalized Bron–Kerbosch algorithm with THRESHOLD = 0 to the graph in Fig. 1, is displayed in Table VI. In this table it is assumed that the procedure ANY MEMBER IN selects the numerically smallest member of its argument. The reader will find it profitable to work out for himself a corresponding table for THRESHOLD = ∞. He will find that it is a little longer to write out but much easier to compute than the THRESHOLD = 0 case.

5. EFFICIENCY CONSIDERATIONS

Tables IV and V give a clear, complete, and concise description of a family of clique-finding algorithms. If we had a compiler that would accept the language facilities listed in Sec. 2, we could run these programs in their present form. Further, if the hardware of the computer used were ideally suited to the program structures and the data structures involved, these programs would be efficient.

In practice, however, it is advantageous to take into account that, in general, procedure entry and parameter passing lead to considerable execution overheads, particularly when recursive procedures are involved. We will, therefore, examine some simple methods by which the number of procedure calls can be significantly reduced.

Steps to remove the parameters from the procedure EXPAND and to remove the recursion are not considered. Such steps may or may not be advantageous, depending on the language, the compiler, and the computer to be used.

5.1. Modified Expand (Table VII)

First, we recall that the procedure EXPAND is given a value of COMBINATION that it either presents to the procedure USE or extends. Thus, if, in a given application of FIND AND USE ALL CLIQUES, a total of c cliques is found, there will be c calls of EXPAND that simply pass the value of COMBINATION on to the procedure USE. All of these calls can be saved by turning the procedure EXPAND outside-in, i.e., by moving the test for a completed clique from the beginning of the procedure to just before the recursive call. This requires the calculation of INTERSECTION $\cap$ $G[I]$ explicitly in a local variable rather than implicitly in passing the parameter. The only logical change that this makes in the complete process is

Table VII. A More Efficient Form of Procedure EXPAND[a]

		Analysis
G1	**procedure** MODIFIED EXPAND (**value set** COMBINATION, INTERSECTION, POSSIBLE);	n_1
G2	**begin set** S; **integer** SIZE OF S;	n_1
G3	START: CALCULATE S;	$n_1 + n_2$
G4	SIZE OF $S :=$ SIZE OF (S);	$n_1 + n_2$
G5	**if** SIZE OF $S > 0$ **then**	$n_1 + n_2$
G6	**begin integer** I; **set** LOCAL INTERSECTION;	$n_2 + c_2$
G7	**while** SIZE OF $S > 1$ **do**	$n_1 + n_2 + c_1 + c_2 - 1$
G8	**begin**	$n_1 + c_1 - 1$
G9	$I :=$ ANY MEMBER IN(S); $S := S \cap \neg \{I\}$;	$n_1 + c_1 - 1$
G10	SIZE OF $S :=$ SIZE OF $S - 1$;	$n_1 + c_1 - 1$
G11	POSSIBLE := POSSIBLE $\cap \neg \{I\}$;	$n_1 + c_1 - 1$
G12	LOCAL INTERSECTION := INTERSECTION $\cap$ $G[I]$;	$n_1 + c_1 - 1$
G13	**if** LOCAL INTERSECTION = { } **then**	$n_1 + c_1 - 1$
G14	USE (COMBINATION $\cup$ { }) **else**	c_1
G15	MODIFIED EXPAND (COMBINATION $\cup \{I\}$, LOCAL INTERSECTION, POSSIBLE $\cap$ $G[I]$);	$n_1 - 1$
G16	**end**;	$n_1 + c_1 - 1$
G17	$I :=$ ANY MEMBER IN (S); **comment** the only member;	$n_2 + c_2$
G18	COMBINATION := COMBINATION $\cup \{I\}$;	$n_2 + c_2$
G19	INTERSECTION := INTERSECTION $\cap$ $G[I]$;	$n_2 + c_2$
G20	**if** INTERSECTION = { } **then**	$n_2 + c_2$
G21	USE (COMBINATION) **else**	c_2
G22	**begin** POSSIBLE := POSSIBLE $\cap$ $G[I]$; **goto** START **end**;	n_2
G23	**end**;	c_2
G24	**end** MODIFIED EXPAND;	n_1

[a] Note:
1. If CALCULATE S from Table V(b) is used, we may remove the declaration of SIZE OF S in line E2 and the calculation of SIZE OF S in line G4.
2. The variables declared in line G6 need not be declared recursively.
3. n_1, n_2, c_1, and c_2 are defined in (11) to (14).

that the initial value of INTERSECTION is not tested for emptiness. However, we know that the initial value is V; thus, provided $V \neq \{\ \}$, the algorithm retains its correctness. If $V = \{\ \}$, the algorithm is undefined.

Second, we note that in the special case in which the set S has only one member, the use of a recursive call of EXPAND to process it incurs the cost of a call and of a **while** loop, which just loops once. Of course, every set S other than those that are initially empty sooner or later reduces to a single member. We can deal with this special case neatly by using the well-known fact that when the *last* statement of a procedure is a call of itself, this recursive call can be replaced by a jump to the start of the procedure. The call of EXPAND corresponding to the member of S selected last satisfies this condition.

These two changes are incorporated in the procedure MODIFIED EXPAND shown in Table VII. This should *not* be regarded as a different algorithm but simply as a more efficient method of implementing the same algorithm.

The number of times each line of this procedure is executed can be analyzed in terms of

c_1 : the number of cliques found at the call of USE in line G14 (11)

c_2 : the number of cliques found at the call of USE in line G21 (12)

n_1 : the number of nodes reached by a call of the procedure EXPAND (13)

n_2 : the number of nodes reached by a jump to the label START (14)

where the c and n defined in (5) and (6) are given by $c = c_1 + c_2$ and $n = n_1 + n_2$. The analysis is shown in Table VII.

A further savings of at least one procedure call in every call of EXPAND or MODIFIED EXPAND can of course be made by substituting the body of the appropriate version of CALCULATE S for the procedure calls in statements C6 and G3.

6. EVALUATION OF THE ALGORITHMS

In this section we evaluate the algorithms obtained in Secs. 4 and 5 and the published Bron–Kerbosch algorithm. We do not include the Basic Method in this evaluation as it is undoubtedly inferior to the other methods. (Any reader who questions this is invited to apply it to finding the single clique of any complete graph.)

The following abbreviations will be used for the remaining algorithms:

GBK_i : the generalized Bron–Kerbosch algorithm with THRESHOLD $= i$, i.e., Table IV with CALCULATE S from Table V(b);

SBK: the simplified Bron–Kerbosch algorithm, i.e., Table IV with CALCULATE S from Table V(c);

OBK: the original Bron–Kerbosch algorithm as published[4];

A superscript m is used to denote the use of MODIFIED EXPAND in place of EXPAND.

Two sets of experiments are described, the first on small graphs with up to 48 vertices, and the second on larger graphs with up to 1000 vertices.

6.1. Small-Graph Experiments

The purpose of these experiments was to study the behavior of the various algorithms on graphs for which the number of vertices is comparable with the word length of the computer to be used. In these circumstances, the abstract data structure **set** can be represented by the individual bits in a few computer words, and a **graph** can be represented by an array of sets.

The OBK algorithm is not included in these experiments since, for small graphs, the data structures that it uses make it very inefficient. Cheng[7] reports this conclusion from experiments carried out on an IBM System/360 with the Bron–Kerbosch algorithm expressed in terms of different data representations.

The author has implemented the remaining algorithms in ALGOL-60 for ICL 1900 Series computers. A variable of type **set** is declared in the ALGOL program as **real** and is therefore represented in a 48-bit double word. The necessary operations on sets ($\cap$, $\cup$, $\neg$, ANY MEMBER IN, etc.) are provided by a suite of assembly-language routines incorporated in the ALGOL program as **external** procedures. With this representation, the least efficient operations are ANY MEMBER IN and SIZE OF. A variable of type **graph** with $N \leqslant 48$ vertices is declared a real array with N elements. The bit pattern in this array corresponds to the representation of a graph demonstrated in Fig. 1(b).

Three experiments were carried out using these ICL ALGOL-60 programs on a 1906S machine with 0.3 μsec main store. The results are reported in the following sections.

6.1.1. Experiment 1 — Random Uniform-Density Graphs

If P is a percentage, let us define a *Random Uniform Density Graph* on N vertices with density P to be a graph with $P \cdot N \cdot (N-1)/200$ randomly chosen edges. For this experiment, graphs with $N = 48$ and P ranging from 0 to 100 were generated and the cliques of each were found by using 10 versions of FIND AND USE ALL CLIQUES. Because the procedure USE

that we employed simply counted the cliques, the execution times given do not include any input/output operations. The results of this experiment are shown in Table VIII.

From these results we note the following:

1. *Effectiveness of our efficiency considerations*

Comparing column (3) with column (5) and column (4) with column (12) we see that the changes, made as a result of the efficiency considerations discussed in Sec. 5, improve the running times by at least 20%.

2. *Effect of changing THRESHOLD*

In practical terms, the value of THRESHOLD governs the amount of effort that the algorithm puts into selecting the special vertex J in (10). THRESHOLD $= 0$ represents maximum effort and THRESHOLD $= \infty$ represents minimum effort. The payoff for finding good values of J is that the size of the search tree is reduced, as is confirmed by the values of n in Table VIII. However, it does not follow that the time to execute the alborithm is also reduced. Time is saved by reducing the value of n, but extra time is spent at each node in calculating the set S. Examination of columns (5) to (12) reveals that, as the density of the graph increases, the value of THRESHOLD giving the shortest running time decreases. The best time for each density is shown in boldface type. A fuller study using random uniform density graphs of other sizes produced the same behavior. In no case was it advantageous to use a value of THRESHOLD < 2.

6.1.2. Experiment 2 — the Moon–Moser Graphs

Let $K(N_1, N_2, \ldots, N_m)$ denote the graph on N vertices, where $N = \sum_{1 \leqslant i \leqslant m} N_i$, formed by partitioning the vertices into m sets with N_i vertices in the ith set and then connecting every pair of vertices that do not belong to the same partition.

If, for a given value of N, we take $m = (N + 1)/3$, $N_i = 3$ for $i = 1 \cdots m - 1$ and $N_m = 2$, 3, or 4 as appropriate, then we get the graph that has the greatest possible number of cliques for any graph on N vertices. This result was proved by Moon and Moser.[9]

For this experiment, the Moon–Moser graphs for $N = 3k$, $k = 3 \cdots 10$, i.e., $K(3, 3, 3)$, $K(3, 3, 3, 3)$, etc., were generated and the cliques of each were found using just two versions of FIND AND USE ALL CLIQUES, namely, GBK_0^m and GBK_∞^m. This is a sufficient test since consideration of the form of these special graphs shows that the set S in every call of EXPAND has exactly 3 members. Thus,

1. values of THRESHOLD in the range 0 to 2 are equivalent; (15)

Table VIII. Experiment 1 Results[a]

(1)	(2)		(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)
P	c		GBK_0	GBK_∞	GBK_0^m	GBK_1^m	GBK_2^m	GBK_3^m	GBK_4^m	GBK_5^m	GBK_6^m	GBK_∞^m
0	48	n	1	1	1	1	1	1	1	1	1	1
		t	0.017	0.014	0.011	0.011	0.011	0.011	0.011	0.011	0.011	**0.008**
10	89	n	54	75	54	55	58	61	63	65	67	75
		t	0.055	0.044	0.042	0.039	0.038	0.037	0.036	0.035	0.035	**0.033**
30	235	n	227	414	227	229	237	246	263	287	300	414
		t	0.190	0.169	0.152	0.146	0.137	0.130	0.128	0.128	**0.125**	0.140
50	687	n	869	1917	869	880	965	1106	1238	1345	1494	1917
		t	0.697	0.678	0.561	0.524	0.486	0.470	**0.469**	0.476	0.494	0.569
70	3100	n	4444	13484	4444	4495	5000	6414	8120	10145	11272	13484
		t	3.458	4.311	2.777	2.625	2.385	**2.351**	2.536	2.902	3.121	3.555
90	32390	n	39151	204165	39151	39167	44249	121185	198050	202376	203370	204165
		t	32.583	61.363	25.621	24.634	**21.206**	32.428	46.522	47.353	47.554	47.735
100	1	n	48	48	48	48	48	48	48	48	48	48
		t	0.095	0.013	0.089	0.009	0.009	0.009	0.009	0.009	0.009	0.009

[a] Note: Times are stated in seconds. The times in column (12) reduce by about 10 % if EBK^m is used rather than GBK_∞^m.

2. values of THRESHOLD in the range 3 to ∞ are quivalent. (16)

The results from this experiment are shown in Table IX.

Form these results we note

1. *Relative speed of the algorithms.* We would expect $GBK_\infty{}^m$ to be better than $GBK_0{}^m$ since all the effort used by $GBK_0{}^m$ to look for the best special vertices J is in vain. The results show that $GBK_\infty{}^m$ is indeed over 30% faster than $GBK_0{}^m$.

2. *The time per clique.* Is it reasonable to expect the time per clique, t/c, to be constant for these or for any other tree search–type clique-finding algorithms? We can cast some light on this matter by considering the analysis of FIND AND USE ALL CLIQUES given in Table IV. Let us take the simplest case, namely the SBK algorithm, and let us also assume that the execution time for each of the set operations $=$, $\neq$, $\cap$, $\cup$, $\neg$, and ANY MEMBER IN is constant, i.e., independent of the operand(s). Under these circumstances, it is easy to see that the total execution time t may be expressed in the form

$$t = t_1 n + t_2 c + t_3 \tag{17}$$

where t_1, t_2, and t_3 are constants. Thus, in order to get t/c to be constant, we require

$$t_3 = 0 \tag{18}$$

and

$$n = \alpha c \tag{19}$$

Table IX. Experiment 2 Results[a]

		Algorithm $GBK_0{}^m$		Algorithm $GBK_\infty{}^m$	
N	k	t (sec)	t/c (μsec)	t (sec)	t/c (μsec)
9	3	0.010	370	0.007	259
12	4	0.031	383	0.021	259
15	5	0.095	391	0.065	267
18	6	0.288	395	0.195	267
21	7	0.880	402	0.592	271
24	8	2.654	405	1.783	272
27	9	7.746	394	5.263	267
30	10	23.139	392	15.745	267

[a] Note: See (20) and (21) for values of c and n.

where α is a constant. Now t_3 is negligible; thus (18) is almost satisfied, but in general (19) is not satisfied. Further, on many computers it is difficult to implement all of the set operations so that their execution times are constant. Thus, in most circumstances, it is not reasonable to expect t/c to be constant.

In the special case of the Moon–Moser graphs of size $N = 3k$, however, we have that

$$n = 1 + 3 + 3^2 + \cdots + 3^{k-1} = \tfrac{1}{2}(3^k - 1)$$

which since

$$c = 3^k \tag{20}$$

gives

$$n = \tfrac{1}{2}(c - 1) \tag{21}$$

Thus (19) is almost satisfied. Note, too, that n is independent of THRESHOLD, which is unusual. So, subject to the behavior of the set operations, we can expect t/c to be constant in this case.

A careful analysis of CALCULATE S and MODIFIED EXPAND shows that the above arguments still hold for the Moon–Moser graphs even when the GBK_i and GBK_i^m algorithms are used.

Looking at the experimental results in Table IX, however, we find that t/c is not quite constant. This is because the execution time for the operation ANY MEMBER IN in this implementation is not constant.

6.1.3. Experiment 3 — Relabeled Graphs

This experiment is designed to illustrate the effect of relabeling the vertices of the graph. These data were derived from an examination timetable problem with 31 examinations. The results with the vertices labeled in (1) ascending order of degree and (2) descending order of degree are shown in Table X.

Consideration of the algorithms shows that labeling the vertices in descending order of degree should be beneficial to the algorithm with THRESHOLD $= \infty$ but that the order should have little significance for low values of THRESHOLD. These conclusions are confirmed by the results obtained.

6.1.4. Conclusions from Small-Graph Experiments

In so far as the experiments described are a fair test for these algorithms, it seems reasonable to draw the following conclusions:

1. For general-purpose clique finding, where the densities of the graphs involved are unknown or may vary widely, the generalized Bron–Kerbosch algorithm with THRESHOLD $= 2$ is best.

Table X. Experiment 3 Results

$N = 31, c = 421$	Ascending degree		Descending degree	
Algorithm	n	t (sec)	n	t (sec)
MBK_0	275	0.201	281	0.210
MBK_1	275	0.198	281	0.203
MBK_2	277	0.194	287	0.195
MBK_3	288	0.189	289	0.187
MBK_4	293	0.188	290	0.183
MBK_5	353	0.193	310	0.181
MBK_6	296	0.172	316	0.158
MBK_∞	579	0.205	341	0.151

2. When the densities of the graphs involved are known to be "low" the simplified Bron–Kerborsh algorithm is best. The interpretation of the word "*low*," however, may vary considerably from one implementation to another. This is illustrated by Table XI, which shows the best values of THRESHOLD for a range of graph densities using the programs discussed above and using similar programs in SAIL[13] and run on a PDP-10 computer. Among the differences that contribute to this change are

(a) the PDP-10 has a single machine instruction to find the left-most 1-bit in a word, thus the operation ANY MEMBER IN can be implemented very efficiently; and

(b) the SAIL language provides operations corresponding to $\cap$, $\cup$, and $\neg$ that operate on a complete word; thus these operations can be performed without incurring the overhead of a procedure call.

On the 1906S, therefore, "low" means "under about 20%" whereas on the PDP-10 it means "under about 70%." On other machines, like the CDC 7600

Table XI. Density *P* vs. THRESHOLD

P	10	20	30	40	50	60	70	80	90
THRESHOLD on ICL 1906S	∞	∞	6	5	4	3	3	2	2
THRESHOLD on PDP-10	∞	∞	∞	∞	∞	∞	∞	3	2

and the Burroughs 6700, which have machine instructions for counting the number of 1-bits in a word, the set operation SIZE OF can be implemented more efficiently than on either the 1906S or the PDP-10. This would tend to reduce the optimum value of THRESHOLD for a given density.

A reader who plans to use a clique-finding algorithm extensively might find it beneficial, before selecting a particular algorithm, to do an experiment similar to Experiment 1 on his own machine, with graphs of the type that arise in his particular application.

6.2. Large-Graph Experiments

The author has been unable to find any report of tests on clique-finding algorithms for graphs with more than about 50 vertices. There are, however, applications that give rise to much larger graphs than this; therefore it was felt worthwhile to conduct further experiments to compare the applications of algorithms discussed in this paper to large graphs with the original Bron–Kerbosch algorithm and to explore the ranges of graph size and density that can reasonably be attempted with these algorithms.

In order to use the algorithms to deal with larger problems, it was necessary to reconsider the representation of the data types **set** and **graph**. The author chose simply to extend the method used for small problems by representing a set as a suitably long array of machine words. For certain sets, this representation is augmented by two integer variables pointing, respectively, to the first and the last nonempty words in the array representing the set. A **graph** is then represented by a two-dimensional array of words with the ith row corresponding to the set $G[i]$. To avoid having to pass array parameters by value, a version of EXPAND with no parameters was used in these implementations. We will use the superscript l to denote these large graph versions of the algorithms.

To provide a direct comparison, OBK was also implemented. This was done because, for large graphs, the data structures used by the original algorithm become more satisfactory than they are for small graphs. One change had to be made, however. Bron and Kerbosch use a **boolean array** called *connected* to represent the graph, but this uses N^2 computer words for a graph with N vertices. To avoid excessive storage requirements for large graphs, this array was packed, thus giving the same representation for a graph as in the other algorithms. All other parts of the algorithm were left unchanged.

Three experiments were carried out using SBK^l, $\text{GBK}_2{}^l$, and OBK programmed in SAIL and run on a PDP-10. The results are reported in the following sections.

6.2.1. *Experiment 4 — Constant Density, Variable Size*

For this experiment, random uniform-density graphs with fixed density $P = 5\%$ and variable size N, ranging from 100 to 1000, were generated and their cliques found using each algorithm. The results are presented in Fig. 2 in the form of curves showing the ratios of the run times obtained. It shows that, for these graphs, the SBK^l algorithm is best, being from 3.3 to 6.7 times faster than OBK and about 13% faster than GBK_2^l. The difference in speed between OBK and the other algorithms is of course almost entirely due to the different data structures used. The tree search that it conducts on a graph with 5% density is not significantly different from the searches conducted by the other algorithms.

6.2.2. *Experiment 5 — Variable Density, Constant Size*

This experiment was similar to Experiment 4 except that the graphs used had fixed size $N = 100$ and the density P was varied from 5% up to about 50%. For all three algorithms, graphs with densities above about 50% fall in the "unreasonable" domain discussed in Sec. 6.2.3. The results obtained are presented in Fig. 3. From these we note that

1. SBK^l is the fastest algorithm for densities of up to about 25%;
2. GBK_2^l is the fastest algorithm above 25% density;

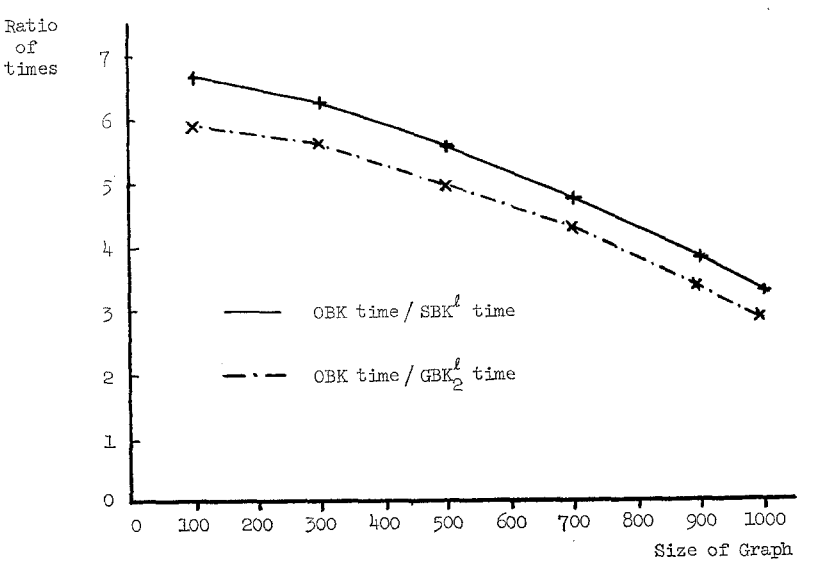


Fig. 2. Results of Experiment 4.

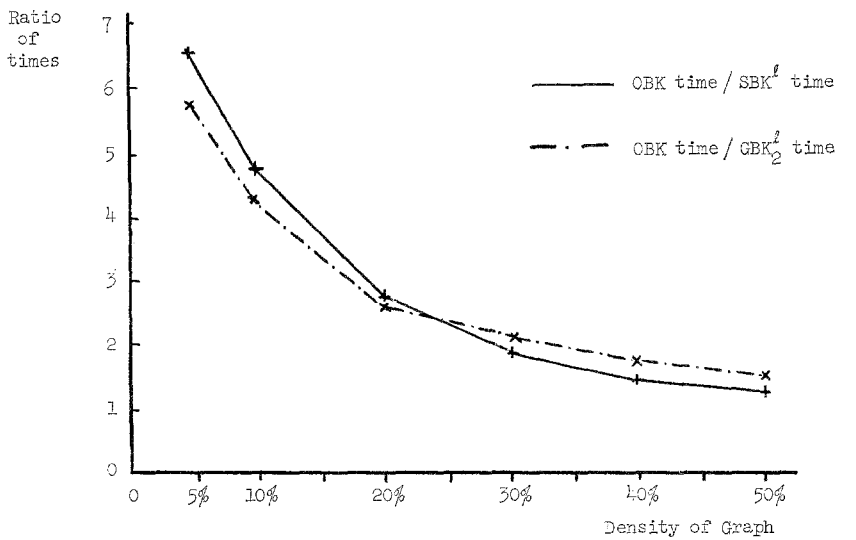


Fig. 3. Results of Experiment 5.

3. the better of SBK^l and $GBK_2{}^l$ is from 1.6 to 6.7 times faster than OBK;
4. for high-density graphs of this size, OBK may become as fast as or faster than the other algorithms.

6.2.3. *Experiment 6 — The "Reasonable" Domain*

Let us arbitrarily define the computation of all the cliques of a graph to be "reasonable" if we can complete it in a few minutes of computer time; otherwise it is "unreasonable." For example, extrapolating from the results in Table IX for the $GBK_\infty{}^m$ algorithm applied to the Moon–Moser graphs, we get

$$N = 33 \qquad t = 0 \text{ min } 48 \text{ sec}$$
$$N = 36 \qquad t = 2 \text{ min } 24 \text{ sec}$$
$$N = 39 \qquad t = 7 \text{ min } 12 \text{ sec}$$
$$N = 42 \qquad t = 21 \text{ min } 36 \text{ sec}$$

Thus, for the machine concerned, we would conclude that the computation is reasonable up to about $N = 36$. If we change to a machine that is 10 times faster, or to one that is 10 times slower, or if we increase our time limit to half an hour, we will not significantly change this conclusion.

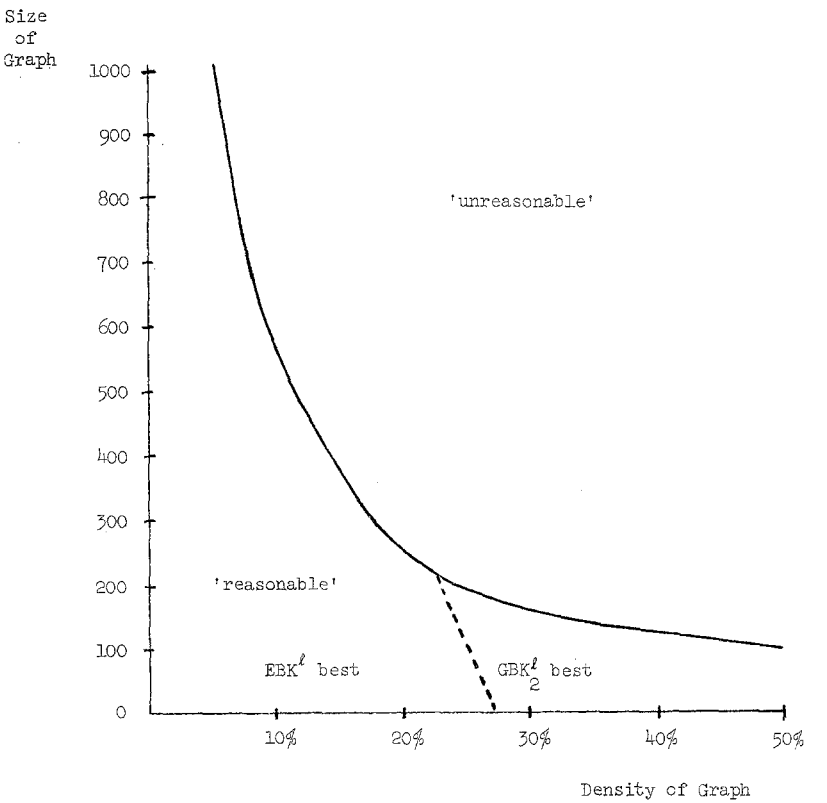


Fig. 4. Results of Experiment 6.

In order to discover the domain of reasonable problems, Experiment 4 was repeated with different fixed densities P and, for each P, the size N was varied over as wide a range as possible from $N = 100$ upwards.

The results obtained are summarized in Fig. 4. This shows the "frontier" between reasonable and unreasonable problems and indicates that, for densities under 10%, we can expect to process graphs with $\mathcal{O}(1000)$ vertices in a reasonable time. For higher densities, the curve falls rapidly until, for densities above about 50%, even a graph with $\mathcal{O}(100)$ vertices requires an unreasonable time. The position of this frontier will not move far with changes of detail in the algorithm or by using a different computer. Earlier experiments that the author carried out on the 1906S computer gave substantially the same curve.

Figure 4 also shows a boundary between the region of the reasonable domain in which the SBK^l algorithm is the best and the region in which it is beneficial to use the more complex $\mathrm{GBK}_2{}^l$ algorithm. However, the position

of this boundary is likely to vary considerably from one machine to another, for the reasons discussed in Sec. 6.1.4. There is no region of the reasonable domain in which the OBK algorithm is best.

7. COMPARISON WITH OTHER NEW CLIQUE-FINDING ALGORITHMS

Augustson and Minker[3] reported in 1970 that the Bierstone algorithm[11] was then the most efficient algorithm. Bron and Kerbosch[4] and Mulligan[10] have since reported that the Bron–Kerbosch algorithm is superior to the Bierstone algorithm. In Sec. 6 we have established that the simplified Bron–Kerbosch algorithm and the generalized Bron–Kerbosch algorithm represent a further improvement over the original Bron–Kerbosch algorithm. It remains to discuss some other clique-finding algorithms that have been published recently. Four such algorithms are discussed briefly in the following sections. In this discussion we use the term "the BK algorithms" to refer collectively to the algorithms of Bron–Kerbosch type.

7.1. The Akkoyunlu Algorithm

Although his approach is very different, the algorithm that Akkoyunlu[1] describes[2] can be regarded as a depth first tree search fundamentally equivalent to the BK algorithms. In Step 3 of the algorithm a member x is chosen at random from a set S_k. This corresponds exactly to the choice of the special vertex J in the BK algorithms. If x is chosen suitably, Akkoyunlu's algorithm can be made to produce the same tree and to generate the cliques in the same order as any one of the BK algorithms.

The major disadvantage is that, to quote Akkoyunlu, "the algorithm manipulates symbolic expressions." A typical expression, which arises in the application of the algorithm to the example graph in Fig. 1, is

$$E(8) \cap L(4, 5) \cap L(5) \cap L(6, 7) \cap L(5, 7, 9)$$

Even with a very good method for representing and operating on such expressions within the computer, it seems unlikely that this algorithm could be implemented to run as rapidly as the BK algorithms. Akkoyunlu does not report any experimental results.

7.2. The Ambler *et al.* Algorithm

Ambler *et al.*[2] describe a clique-finding algorithm very briefly. In private communications with Barrow, the author has established that this

[2] Akkoyunlu's published algorithm contains a typographical error; the set J defined in Step 5 on p. 6 should read $J = \{j \mid j \in I, \quad x \notin S_j\}$ rather than $J = \{j \mid j \in I, \quad x \in S_j\}$.

algorithm and various more recent unpublished developments of it are also fundamentally similar to the BK algorithms but are not more efficient than the simplified and generalized Bron–Kerbosch algorithms.

7.3. The Das Algorithm

In order to find the cliques of a graph G, the Das algorithm[8] operates on a list of the edges of the complementary graph $\bar{G}$. Das calls the list an "edge inclusion table." By performing a binary tree search on this data structure, the algorithm generates sets of vertices that Das calls "irredundant solutions." Each irredundant solution of $\bar{G}$ is the complement of a clique of G.

The disadvantage in this algorithm is that it initially generates some "redundant solutions" of $\bar{G}$, which correspond to submaximal completely connected sets of G. Thus, potential solutions are stored as they are found, and the final step of the algorithm is to check through the complete list and delete the redundant ones. This is reminiscent of some of the older clique-finding algorithms and is extravagent in the use of both store and time.

Das mentions that his algorithm is "readily programmable on an IBM 360 system by using APL," but does not quote experimental results. It would certainly not be as efficient as the BK algorithms.

7.4. The Osteen–Tou Algorithm

The algorithm described by Osteen and Tou[12] is yet another depth first tree search. Like the Das algorithm it has the disadvantage that it requires the simultaneous storage of all the cliques, since it is expressed as a procedure that delivers the complete set of cliques as a result. For graphs with a large number of cliques it therefore uses a large amount of store.

Also, in order to prevent the generation of submaximal, completely connected sets, the algorithm includes a special check (Step 8) which, while it is not as time consuming as that required in the Das algorithm, is a cumbersome way of avoiding incorrect solutions.

Osteen and Tou present limited experimental results obtained using PL/1 on an IBM System 360/65. The times they quote for finding the cliques of small Moon–Moser graphs are 60 to 70 times larger than the times given in Table IX. A difference of this magnitude cannot be attributed totally to the difference in machines and languages used.

7.5. Summary

While the above discussion is not as satisfactory as a direct comparison of actual implementations on the same computer, it seems reasonable to

conclude that none of the algorithms discussed is potentially better than the BK algorithms.

8. CONCLUSIONS

Our conclusions relate first to the methods by which the algorithms have been developed and second to the algorithms themselves.

8.1. Methods of Development

The technique of program evolution used, a variation on normal stepwise refinement, is a powerful tool for the development of algorithms, particularly in the field of combinatorial programming. Its main advantages are the following:

1. The changes made to an algorithm at each step can be sufficiently straightforward to be fully understood and perhaps rigorously proved.

2. The use of abstract data structures allows the choice of concrete representations to be postponed until the algorithm is complete. The operations to be performed on the data structures are then known; thus, representations best suited to these operations, and to the demands of a particular application, can be selected.

3. Processes like the removal of parameters and/or recursion, to save space and execution overheads, and the introduction of redundant data, to save computation, can be considered at an advanced stage of the evolution process if necessary.

4. The development of algorithms related to a given algorithm can be achieved by starting at an appropriate point in the evolution and evolving in a different direction. For example, in the case of cliques, an algorithm to count the cliques of a graph or one to find the largest clique can easily be produced.

8.2. The Algorithms

There are two main conclusions:

1. For applications that give low-density graphs, the simplified Bron–Kerbosch algorithm is best, while for higher densities it becomes worthwhile to use the generalized Bron–Kerbosch algorithm with a value of THRESHOLD $\geqslant 2$. The precise interpretation of the terms "low" and "higher" depends on the details of the implementation and on the hardware of the computer used.

2. The largest graphs that can be processed in a reasonable time by these algorithms are of $\mathcal{O}(1000)$ vertices for low densities and $\mathcal{O}(100)$ for high densities.

ACKNOWLEDGMENTS

The author has pleasure in thanking W. D. Davison and C. A. R. Hoare for their helpful discussions, H. G. Barrow, E. W. Dijkstra, and the referees for their valuable written comments, and D. E. Knuth for access to his file of literature on this subject.

REFERENCES

1. E. A. Akkoyunlu, The enumeration of maximal cliques of large graphs, *SIAM J. Computing* **2**:1–6 (March 1973).
2. A. P. Ambler, H. G. Barrow, C. M. Brown, R. M. Burstall, and R. J. Popplestone, A versatile computer controlled assembly system, *Third International Joint Conference on Artifial Intelligence: Advance Papers* (August 1973), pp. 298–303.
3. J. G. Augustson and J. Minker, An analysis of some graph theoretical cluster techniques, *J. ACM* **17**:571–588 (October 1970).
4. C. Bron and J. Kerbosch, "Algorithm 457: Finding all cliques of an undirected graph," *Commun. ACM* **16**:575–577 (September 1973).
5. C. Bron, J. Kerbosch, and H. J. Schell, "Finding Cliques in an Undirected Graph," Technical Report, Technological University of Eindhoven, The Netherlands.
6. R. M. Burstall, "Tree searching methods with an application to a network design problem," *Mach. Intell.* **1**:65–85 (1967).
7. C.-M. Cheng, "Clustering by Clique Generation," M.Sc. thesis, Department of Computer Science, University of Illinois, Urbana, Illinois (June 1974).
8. S. R. Das, "On a new approach for finding all the modified cut-sets in an incompatibility graph," *IEEE Trans. Comput.* **C-22**:187–193 (February 1973).
9. J. W. Moon and L. Moser, "On cliques in graphs," *Isr. J. Math.* **3**:23–28 (1965).
10. G. D. Mulligan, "Algorithms for Finding Cliques of a graph," Technical Report No. 41, Department of Computer Science, University of Toronto, Ontario, Canada (May 1972).
11. G. D. Mulligan and D. G. Corneil, "Corrections to Bierstone's algorithm for generating cliques," *J. ACM* **19**:244–247 (April 1972).
12. R. E. Osteen and J. T. Tou, "A clique-detection algorithm based on neighborhoods in graphs," *Int. J. Comput. Inf. Sci.* **2**:257–268 (1973).
13. K. A. Van Lehn, "SAIL User Manual," Technical Report STAN-CS-73-373, Computer Science Department, Stanford University, California (July 1973).