An ALGOL 68 package for implementing graph algorithms*

G. R. Garside and P. E. Pintelas†

Computing Laboratory, University of Bradford, West Yorkshire, BD7 1DP

The implementation of graph-theoretic algorithms using the facilities of standard algorithmic languages is not easy since data structures and operations natural to the subject are not readily available. GRAAP (GRaph Algorithmic Applications Package) is a new system designed to solve this problem. Written in ALGOL 68-R it consists of about 150 operators and procedures which perform operations natural to graph theory and essential to the implementation of graph algorithms. These operators and procedures manipulate information representing graphs and related objects stored in suitably defined structures. GRAAP exists as an album of precompiled segments to minimise compilation time. The operations provided and the transparent internal representations of graphs of different kinds are discussed. The ease with which algorithms can be implemented is demonstrated by examples.

(Received January 1979)

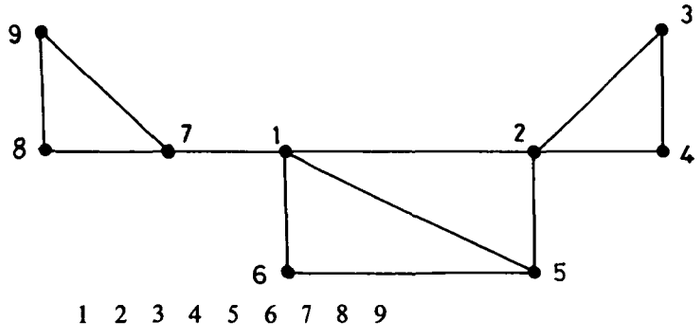
1. Introduction

A graph can be used to represent many physical situations which involve discrete objects and relations between them. The objects are represented by the nodes of the graph and the relations between them are represented by the edges of the graph. We do not repeat here the terminology associated with graphs as this can be found in the many texts now available on the subject. Two such are Deo (1974) and Christofides (1975); these also provide excellent accounts of the diverse applications of graphs. The solution to a problem whose fundamental nature can be represented by a graph may often be obtained by manipulating the graph in a number of discrete steps according to some algorithm. We shall refer to such algorithms as *graph algorithms*. Example 3.1 implements an algorithm for finding the cliques of a graph using the package described in this paper. In particular the cliques of the graph of **Fig. 1** are found (see **Fig. 5**).

Attempting to implement graph algorithms on a computer raises a number of programming problems. These involve the efficient representation and manipulation of a wide variety of graphs of different types and complexity. It is desirable to have facilities which directly use the objects and operations natural to the subject and this has been the aim of the graph algorithmic applications package (GRAAP) described in this paper. The package is written in ALGOL 68-R for use on ICL 1900 series machines. It should be able to cater for most needs immediately but the existing facilities can be used as a basis from which to extend into the user's area of application. This potential extensibility of GRAAP makes it unlike any other package or language so far produced in the area of graph algorithms. The package, which includes structures for representing graphs and related objects as well as routines for handling them, is available as an ALGOL 68-R segmented album and so a user's program need only include those segments specifically required for the implementation of his algorithm.

The object of this paper is to outline the package and its uses; full details are given in Garside and Pintelas (1978). We first sketch the background against which GRAAP has been developed.

A large number of software aids for implementing graph algorithms already exist but they are all much more limited than GRAAP. Some of the more important of these are mentioned below. Standard algorithmic languages such as FORTRAN or ALGOL 60, with their restricted data structures, are unsuitable for implementing graph algorithms, while list processing languages provide more appropriate data structures but tend to hide the graph-theoretic nature of the algorithm and lead to slow execution and large demands for storage (see Rheinbolt, Basili and Mesztenyi, 1972). Special purpose languages and software systems have been developed to free the programmer from representational problems and assist him with the manipulation of graph structures. It would be inefficient to design a new language and write a compiler for it because of the limited application of the language and the enormous amount of time required to produce the compiler. Consequently all graph-theoretic languages are embedded in some well known high level language. There are two main approaches. The first is to extend the host language by designing extra language constructs to take care of the graph statements and expressions. This requires a preprocessor which translates the graph statements into statements of the host language. Languages using this approach are GTPL of King (1970) which is based on FORTRAN II, GEA of Crespi-



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

Adjacency matrix

NODE 1 [2 5 6 7]
NODE 2 [1 3 4 5]
NODE 3 [2 4]
NODE 4 [2 3]
NODE 5 [1 2 6]
NODE 6 [1 5]
NODE 7 [1 8 9]
NODE 8 [7 9]
NODE 9 [7 8]

Adjacency lists

Fig. 1

*This work was supported by the NATO Science Fellowship Programme.

†Now at DACC, Technical Operations Department, 24 Strat. Syndesmou, Athens, Greece.

Downloaded from https://academic.oup.com/comjnl/article/23/3/237/375193 by guest on 09 April 2024

Reghizzi and Morpurgo (1968) which is an ALGOL 60 extension with digraphs as a new data type, GASP at the University of Illinois (Chase, 1970) which is an extension of PL/I and GRASPE at the University of Texas (Friedman, 1968) which is an extension language for processing digraphs and has been embedded in SNOBOL4, SLIP-FORTRAN and LISP 1.5. For a full account of existing graph processing aids the reader is referred to Pintelas (1976). The second approach requires the host language to possess facilities which enable new data structures and operations to be defined. ALGOL 68 is such a language and we have used an implementation of this, ALGOL 68-R on ICL 1900 series computers, in producing GRAAP. As far as we know, no other package is available which uses a language with these kinds of facilities. Consequently GRAAP has much wider application than any of the above-mentioned extensions and has already been used to implement many existing graph algorithms and to develop new ones in various fields.

Section 2 describes the GRAAP structures and some of the routines for handling them, while two examples of the use of the package are given in Section 3. In conclusion Section 4 contains observations based on the experience of using the package extensively.

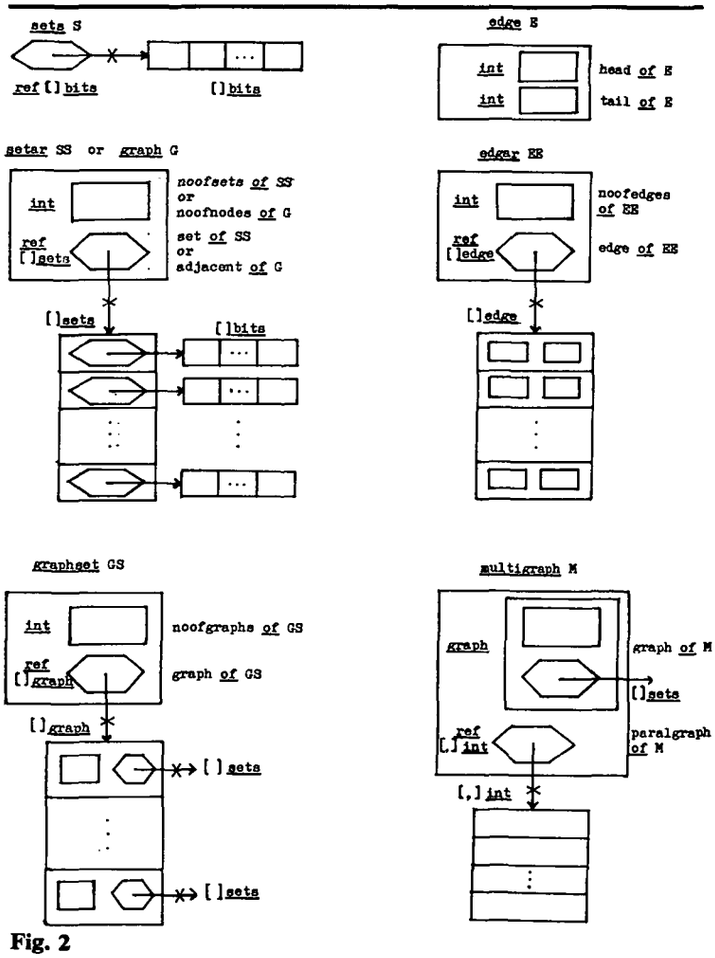


Fig. 2

2. Construction of GRAAP

In order to decide which facilities to provide in GRAAP a large number of existing graph algorithms were investigated, on the basis that the facilities required by these algorithms would be likely to be those required by future algorithms.

There are many ways in which graphs and related objects can be represented (see Deo (1974) and Pintelas (1976)) and the efficiency of implementation of a particular graph algorithm depends upon the representation chosen. It was not practical to include all the various representations in one package, being better to choose one which would be optimal for a large number of algorithms and adequate for many others. Thus the structures in GRAAP are based upon the bit representation of sets and adjacency matrices of graphs. In some cases this may not lead to the most efficient representation but the large number of operations available in the package for manipulating the structures should more than compensate for this. Some facilities are provided for representing and handling graphs using their edge listings and incidence matrices.

In order to allow maximum flexibility in the structures used to represent sets and graphs, a number of global objects of

mode setsbound = struct (int *bound*)

have been declared specifically to enable the structures to be of user defined sizes. The mode declarations of the GRAAP structures are given in Section 2.1 and the storage allocation is shown in **Fig. 2** in which the actual storage beyond a crossed line is generated on the heap by user calls to procedures whose names are given in **Table 1.**

2.1 *GRAAP structures*

2.1.1 *Representation of sets*

A convenient and efficient representation of an unordered set of non-negative integers is by means of a string of binary digits, e.g. the set $\{0, 1, 4, 6\}$ from the universe $\{i \mid 0 \leqslant i \leqslant 10\}$, say, can be represented by the eleven-bit string 11001010000. To implement this representation in GRAAP we declare

mode sets = ref [ ] bits

A row of sets is declared as

mode setar = struct (int *noofsets*, **ref [ ] sets** *set*)

This has proved useful as a stack for holding subsets and partitions of a set. The variable *noofsets* is analogous to a stack pointer in indicating the last meaningful bit string in a **setar** object.

When storage is generated for a **sets** object or a **setar** object, the user defined variables *bound* **of** *setsbound* and *bound* **of** *setarbound* determine the actual amount of storage generated (see Table 1). In Example 3.1 (see Fig. 5) *bound* **of** *setsbound*, *bound* **of** *setarbound* and *bound* **of** *noofnodes* (see Section 2.1.2),

Table 1

Structure	*User-valued global variables required for storage generation*	*Value to be assigned*	*Storage generation procedure*
sets	*bound* **of** *setsbound*	Largest element allowed in a set	*genset*
setar	*bound* **of** *setarbound*	Maximum number of sets allowed in an array of sets	*gensetar*
graph	*bound* **of** *noofnodes*	Maximum allowable index of a node of a graph	*gengraph*
edgar	*bound* **of** *graphedges*	Maximum number of edges allowed in a graph	*genedgar*
multigraph	*bound* **of** *paraledges*	Maximum number of parallel edges allowed between any two nodes in a multigraph	*genmlgraph*
graphset	*bound* **of** *graphbound*	Maximum number of graphs allowed in an array of graphs	*gengraphset*

Downloaded from https://academic.oup.com/comjnl/article/23/3/237/375193 by guest on 09 April 2024

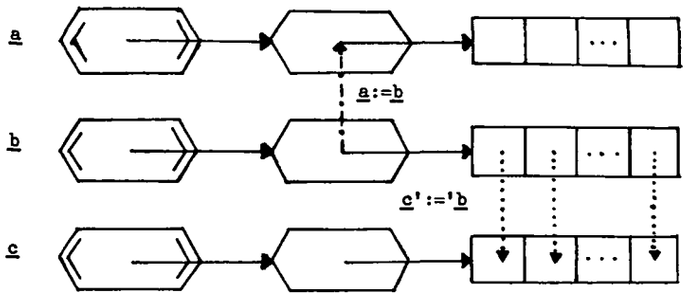


Fig. 3

all of mode **setsbound,** are given user defined values through integers read in and assigned. For the graph of Fig. 1, these took the values 9, 27 and 9 respectively, 9 being the number of nodes and 27 the maximum possible number of cliques calculated from the formula given by Even (1973, p. 154), which is $3^{n/3}$ when the number of nodes $n \equiv 0$ (*mod* 3).

2.1.2 *Representation of graphs*

There are many possible ways of representing graphs and GRAAP provides two explicit methods for both directed and undirected graphs. The first is based on the adjacency matrix which assumes a numbering of the nodes from one upwards and is defined as $[a_{ij}]$ where $a_{ij} = 1$ if node i is joined by an edge to node j and 0 otherwise. The structure provided for this representation is similar to **setar** and is declared as

mode graph = **struct** (**int** *noofnodes*, **ref** [ ] **sets** *adjacent*)

However, **graph** is more restricted than **setar** since both the number of bits in each set and the length of the row of sets are determined by the value of *bound* **of** *noofnodes* when storage is generated (see Table 1). In a declared graph, *G*, the value of *noofnodes* **of** *G* is that of the highest numbered node of *G*. Nonexistent nodes are indicated by a bit string consisting entirely of zeros and isolated nodes have a one in bit position zero.

The second method of representing a graph in GRAAP is by an edge listing. This is the set $\{\langle i, j\rangle\}$ of all the edges in the graph and is implemented through

mode edge = **struct** (**int** *head*, *tail*)
mode edgar = **struct** (**int** *length*, **ref** [ ] **edge** *edge*)

The value in *length* will be the highest index of a meaningful edge in a particular **edgar** object.

A row of graphs can be stored using

mode graphset = **struct** (**int** *noofgraphs*, **ref** [,] **graph** *graph*)

A multigraph, which may have more than one edge between a pair of nodes, is stored as an adjacency matrix and a packed integer array. It is declared as

mode multigraph = **struct** (**graph** *graph*, **ref** [,] **int** *paralgraph*)

and the number of integers packed into a single word of *paralgraph* is determined by the length of the bit string representing the upper limit of the number of parallel edges between two nodes, held in *bound* **of** *paraledges* (see Table 1).

2.2 *GRAAP operators and procedures*

The number of operators and procedures provided to manipulate values held in structures described in Section 2.1 are too numerous to list here, there being nearly 150 available to the user. Full descriptions, including priority numbers, are contained in Pintelas (1976) and Garside and Pintelas (1978). The object of this section is to outline some of the more useful ones, including all those which are used in the two examples of Section 3, and these are grouped below under facility headings.

2.2.1 *Input/output*

Procedures are provided for reading into and printing out from all the structures previously described. In Example 3.1 the procedures *readgraph*, *printgr*2 and *printsetar* are used. *readgraph* reads graphs as adjacency lists and stores the information in objects of mode **graph**; *printgr*2 prints the graphs in adjacency list form and *printsetar* prints each **setar** object in its parameter list as a list of sets (see Fig. 5).

2.2.2 *Initialisation*

The operator **clear** sets all appropriate locations to zero when applied to objects of mode **setar, graph, edgar** and **multigraph. nulset** generates a bit string consisting entirely of zeros (the empty set) , whilst **mulval** *n* (see Example 3.2, **Fig. 7**) generates a set with 1's in bit positions 0 to *n*.

2.2.3 *Assignment*

With a few exceptions the operator ':=' must be used in preference to the normal ALGOL 68 assignment symbol := for all assignments involving those structures introduced in Section 2.1 whose modes contain references. This is because the normal assignment symbol will merely copy the reference instead of all the information required. As an example, consider the following code:

sets *a*, *b*, *c*; *genset*((*a*, *b*, *c*)); *readset*(*b*); *a* := *b*; *c* ':=' *b*;

The result is shown in **Fig. 3.** The full lines show the pointers before the assignments and the dotted lines show the transfer of information. The line of dots and dashes shows how the reference pointer for *a* has been changed to point to the [ ] **bits** of *b*. This means that any subsequent change to *a* or *b* only changes the [ ] **bits** of *b*.

A similar argument holds for assignments involving other structures with references. Thus *A* ':=' *B* where *A* and *B* have the same mode copies to *A* all those parts of *B* which are not references. In the case of *A* being of mode **ref setar** and *B* of mode **ref sets,** the [ ] **bits** of *B* is copied to the [ ] **bits** of (*set* **of** *A*) [*noofsets* of *A* + 1] and *noofsets* **of** *A* is increased by 1. An example of this can be seen in the assignment statement which forms the body of the inner loop of the program of Example 3.1 (see Fig. 5). Here *A* is the **ref setar** object *S* and *B* is the **ref sets** object produced by the complicated expression to the right of the ':=' symbol.

2.2.4 *Natural operations*

These are the operations which are 'natural' to particular set-theoretic and graph-theoretic entities. In the case of sets we include, among others, union, intersection and cardinal number. The program of Fig. 5 contains the operators (i) **set,** which inserts an element in a set, (ii) *, which is the set intersection operator and (iii) **single,** which generates a singleton set.

In the procedure *bicon* of Fig. 7, the need to control a loop by running through the adjacency list of node *v* in a graph *G* is effected by

w **startat** 1; **while** *w* **elof** (*adjacent* **of** *G*) [*v*] **do** *loop*

where **startat** initialises the controlled variable *w* and **elof** produces the next element in the adjacency list, returning **true** if the list is not exhausted.

For graphs, most of the natural operations are provided as procedures, e.g. insert an edge, find the complement of a graph, form a subgraph. In Fig. 7 *noofedges*(*G*) is used to set the upper limit of the edge stacks by returning the number of edges in the graph *G*.

2.2.5 *Boolean operations*

The facility to perform tests using the new structures is provided. Some examples are given in **Table 2** below, in which *i*

Downloaded from https://academic.oup.com/comjnl/article/23/3/237/375193 by guest on 09 April 2024

Table 2

Boolean expression	*Test being made*
i **elem** *S*	does $i \in S$?
T **sub** *S*	is $T \subseteq S$?
i **isol** *G*	is *i* an isolated node?
e **elem** *E*	does $e \in E$?

is of mode **ref int,** *S* and *T* are of mode **ref sets,** *G* is of mode **ref graph,** *e* is of mode **ref edge** and *E* is of mode **ref edgar.**

2.2.6 *Storage generation*
The statement

graph *G*

declares *G* to be of mode **ref graph.** The details of a particular graph cannot yet be stored since no storage beyond the crossed line for **graph** in Fig. 2 has yet been allocated. So before we can store a graph in *G* we need the statement

gengraph (*G*)

and since the size of storage allocated depends on the global variable *bound* **of** *noofnodes*, it is imperative that we have initialised this before the procedure call. The principle holds for all the structures except **edge** declared in Section 2.1 (see Table 1). The point is well illustrated in Example 3.1, where the procedures *gensetar* and *gengraph* are called and the global variables *bound* **of** *noofsets* and *bound* **of** *noofnodes* have been previously read in.

However, it is not always necessary to generate storage explicitly. In Example 3.2 the operator **mulval** in the declaration

sets *S* := **mulval** *n*

(see Fig. 7) causes [ ] **bits** storage to be generated, filled with 1's and returned. The **ref** [ ] **bits** pointer of *S* is then made to point to the newly generated storage by the action of the assignment symbol.

The GRAAP Reference Manual (Garside and Pintelas, 1978) clearly indicates the occasions when storage should be generated explicitly.

2.2.7 *Advanced procedures*
Many graph algorithms have been easily and successfully implemented using GRAAP. Some of the most widely useful of these have been included in the final version of the package as 'advanced procedures'. Examples of these are procedures to find spanning trees, bridges and cutnodes. In Example 3.1 (see Fig. 5) the procedure *maximalsets* with a **ref setar** argument, *S*, returns a **setar** object which contains precisely those elements of *S* which are not proper subsets of other elements of *S*.

2.2.8 *Algorithm testing*
To aid the development, testing and efficiency investigations of graph algorithms, the facility for generating pseudo-random sets, graphs and multigraphs is provided through procedures such as *randomset*, *randgraph* and *randmlgraph*. In these procedures the user has control over average element and edge densities.

2.2.9 *The segmented album*
GRAAP consists of a set of precompiled segments making up an ALGOL 68-R album. It is possible to use only those segments required in a given applications program, e.g. the first line of Fig. 5 indicates the segments required for Example 3.1.

The four major segments of GRAAP1, the extant version on the ICL 1904S* at the University of Bradford, are shown in **Fig. 4,** together with their dependence on each other (shown as a digraph!). Each advanced operation is placed in its own segment and so must be referenced separately in the segment listing (as is the case with *maximalsets* in Fig. 5). Each advanced operation contains calls to operators and/or procedures in the four main segments.

3. Programmed examples

3.1 *Cliques of a graph*
A *clique* of a graph is a maximal subset, *C*, of the nodes with the property that each node in *C* is adjacent to all other nodes in *C*. The algorithm coded here, which finds all the cliques of a graph, is a modified version of algorithm 8.1 of Even (1973, p. 155), and originally due to Paull and Unger (1959). For practical purposes this algorithm has been superseded and it is included here only because its simplicity makes it an ideal introductory illustrative example. In fact the problem of finding all the cliques of a graph is an *NP-complete* problem and more practical algorithms are considered by Johnston (1976). Johnston develops a family of algorithms based on a method due to Bron and Kerbosch (1973) and presents them in an abstract ALGOL-like language for which he needs special types for sets and graphs, as well as many set-theoretic and graph-theoretic representations and operations. All these requirements are present in GRAAP and the algorithms given by Johnston have been programmed successfully using the package. A more recent paper by Gerhards and Lindenberg (1979) describes two new algorithms for finding all the cliques of a nondirected graph. These are based on sophisticated tree-searching techniques which lead to better computational speeds when compared with the method of Bron and Kerbosch and the algorithms are constructed in such a way as to allow parallel processing. It is intended to implement these algorithms using GRAAP, although the parallel processing facility cannot be utilised.

A program which implements the algorithm given by Even and uses the GRAAP facilities is shown in Fig. 5. The *sets*, *setars* and *graph* segments of the album GRAAP1 together with the advanced procedure *maximalsets* are required. The output produced is also shown in Fig. 5.

The algorithm as given by Even uses three arrays of sets called *S*, *C* and *S'*. The housekeeping provided by the operators and procedures in GRAAP enables us to declare only one **ref setar** object, *S*. In particular the operator ':=' for assigning a **sets** value to a **setar** object automatically updates the *noofsets* pointer and thus implicitly forms the quantity $S \cup C$ of the algorithm in *S* of the program. The procedure *maximalsets* returns a **setar** object consisting of all those sets in its **setar** argument which are not subsets of others. Assigning this returned value to *S* dispenses with the need for *S'* of the

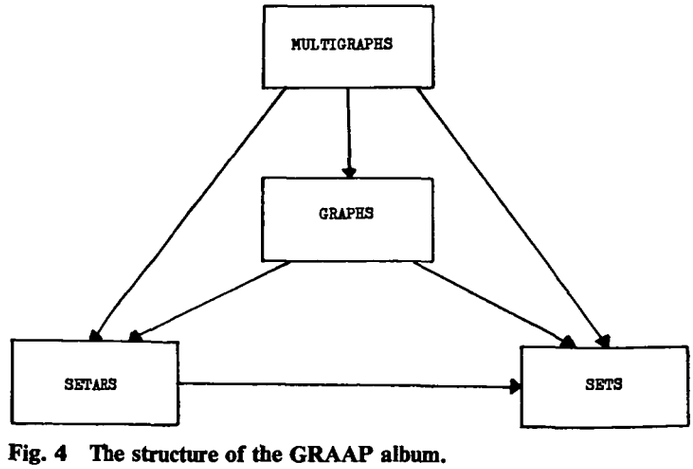


Fig. 4 The structure of the GRAAP album.

Downloaded from https://academic.oup.com/comjnl/article/23/3/237/375193 by guest on 09 April 2024

```
cliques with sets, setars, graphs, maximalsets from GRAAP1
begin
 proc cliques = (graph G) ref setar:
  begin int p;
   ref setar S = setar;
   gensetar(S); clear S; S ':=' single 1;
   for i to noofnodes of G - 1 do
    begin p := noofsets of S;
     for j to p do
      S '=' (i + 1) set ((adjacent of G) [i + 1] * (set of S)
      [j]);
      S ':=' maximalsets(S)
    end;
  S
  end;
 graph G;
 read ((bound of noofnodes, bound of setarbound));
 bound of setsbound := bound of noofnodes;
 gengraph (G);
 readgraph(G); printgr2(G);
 print ((newline, newline, 'the cliques of the graph are'));
 printsetar (cliques(G))
end
finish
```

Output produced with the graph of Fig. 1 as Data

```
GRAPH

NODE 1 [2 5 6 7]
NODE 2 [1 3 4 5]
NODE 3 [2 4]
NODE 4 [2 3]
NODE 5 [1 2 6]
NODE 6 [1 5]
NODE 7 [1 8 9]
NODE 8 [7 9]
NODE 9 [7 8]

THE CLIQUES OF THE GRAPH ARE
[[2 3 4] [1 2 5] [1 5 6] [1 7] [7 8 9]]
```

Fig. 5 GRAAP program for Example 3.1.

```
{(4, 2), (3, 4), (2, 3)}
{(6, 1), (5, 6), (5, 1), (2, 5), (1, 2)}
{(9, 7), (8, 9), (7, 8)}
{(1, 7)}
```

Fig. 6 The biconnected components of the graph of Fig. 1.

algorithm.

Of the other GRAAP operators used in this implementation, **clear** S fills S with empty sets and puts *noofsets* of S to zero, **single** 1 generates the singleton set $\{1\}$, * is the set intersection operator and **set** causes the element indicated on the left to be added to the **sets** object on the right.

3.2 *Biconnected components of a graph*

A biconnected component of a connected graph, G, is a maximal subgraph H, such that there are at least two distinct paths between each pair of nodes in H. The biconnected components of the graph of Fig. 1 are shown as edge listings in **Fig. 6.** If the removal of a node and the edges incident on that node causes the resultant graph to be disconnected then the node is called an articulation point. The articulation points of the graph of Fig. 1 are 1, 2 and 7 and are precisely those nodes which occur in more than one biconnected component.

Tarjan (1972) gives an efficient algorithm for finding the biconnected components of a graph and which is based on the

```
proc biconnect = (ref graph g) ref edgar:
 begin int i, n := noofnodes of g, sp, cp;
  ¢ sp = stack pointer, cp = current component pointer ¢
  edgar ccps; genedgar(ccps); clear ccps;
  [1:n] int number, lowpt; clear number; clear lowpt;
  [1:noofedges(g)] edge stk, ccp;
  ¢ stk = stack, ccp = current component stack ¢
  proc bicon = (int v, u) void:
   begin int w;
    number [v] := i plus 1; lowpt [v] := i;
    ¢{{number [v] := i := i + 1; lowpt [v] := number [v];}¢
    w startat 1; while w elof (adjacent of g) [v] do
    ¢{for w in the adjacency list of v do}¢
    if number [w] = 0 then
    ¢{if w is not yet numbered then}¢
     stk [sp plus 1] := (v, w);
     ¢{add (v, w) to stack of edges}¢
     bicon (w, v);
     if lowpt[v] > lowpt[w] then lowpt[v] := lowpt[w] fi;
     ¢{lowpt(v) := min(lowpt(v), lowpt(w))}¢
     if lowpt[w] ≥ number[v] then
     ¢{start new biconnected component}¢
      if cp ≠ 0 then
       (edge of ccps) [length of ccps plus 1] := (0, 0);
       for i to cp do
        (edge of ccps) [length of ccps plus 1] := ccp[i];
        cp := 0 fi;
       ¢ store previous component in edgar ccps ¢
       while number[head of stk[p]] ≥ number[w] do
       ¢{while top edge = (u1, u2) on edge stack has
         number(u1) ≥ number(w) do}¢
        begin ccp[cp plus 1] := stk[cp]; sp minus 1 end;
        ¢{delete (u1, u2) from edge stack and add
          it to current component}¢
        ccp[cp plus 1] := (v, w); sp minus 1
        ¢{delete (v, w) from edge stack and add
          it to current component}¢
      fi
     else
     if number[w] < number[v] and w ≠ u then
     ¢{if (number(w) < number(v)) and (w ≠ u) then}¢
      stk[sp plus 1] := (v, w);
      ¢{add (v, w) to edge stack}¢
      if lowpt[v] > lowpt[w] then lowpt[v] := lowpt[w] fi
      ¢{lowpt(v) := min(lowpt(v), lowpt(w))}¢
     fi
    fi
   end;
  i := 0; sp := cp := 0; ¢{i := 0; empty edge stack}¢
  sets s := mulval n;
  int w;
  w startat 1; while (w elof s) and number w = 0 do
    bicon(w, 0);
  ¢{for w a vertex do if w is not yet numbered then
    bicon(w, 0)}¢
   if cp ≠ 0 then
    (edge of ccps) [length of ccps plus 1] := (0, 0);
    while cp > 0 do
     begin (edge of ccps) [length of ccps plus 1] := ccp[cp];
           cp minus 1
     end
    ¢ transfer last biconnected component to edgar ccps ¢
   fi;
  ccps
 end
```

Fig. 7 GRAAP program for Example 3.2.

Downloaded from https://academic.oup.com/comjnl/article/23/3/237/375193 by guest on 09 April 2024

depth first search technique. The algorithm is presented in ALGOL-like terms and involves recursion. It is easily implemented using GRAAP and this coding is shown in Fig. 7 which, to emphasise the ease of implementation, includes statements of the original algorithm as comments between the pairs of symbols ¢{ and }¢ alongside the GRAAP source code. Other comments appear between the normal ¢ symbols.

For a graph with N nodes and E edges the algorithm requires $O(N + E)$ time. Table III shows the average time taken to find the biconnected components of a graph on N nodes with edge density ρ (i.e. $N^2\rho 2$ edges) for $N = 10(10)50$ and $\rho = 0{\cdot}3(0{\cdot}2)$ 0·9. Each time is obtained by averaging over 50 randomly generated graphs. The timings confirm that the algorithm has $O(N + E)$ time complexity.

Average time, in seconds, to find the biconnected components of a connected graph with a given number of nodes and edge density.

Table 3

Edge density, ρ	0·3	0·5	0·7	0·9
No. of nodes, N				
10	0·053	0·063	0·076	0·089
20	0·189	0·255	0·305	0·349
30	0·405	0·522	0·649	0·735
40	0·706	0·920	1·125	1·385
50	1·099	1·475	1·784	2·154

4. Conclusions

The package described in the previous sections possesses the following attributes:

1. Extensibility.
2. Transparency of internal structures.
3. Operations which are natural to graph theory.
4. Ability to accommodate large graphs.
5. Easy implementation of graph algorithms.

Experience with the package has shown these attributes to be extremely useful in solving a wide variety of problems. Algorithms which have been implemented include (i) finding spanning trees, (ii) computing transitive closures, (iii) graph colouring, (iv) cycle generation, (v) finding shortest paths, (vi) finding blocks and cutnodes, (vii) solving partitioning problems, (viii) pathfinding in electrical networks, coming from many different scientific disciplines. The algorithms have been easily implemented using the GRAAP facilities and running times show the package to be efficient.

Acknowledgements

The authors wish to thank Roger Ward for his help in implementing the package and Professor R. J. Ord-Smith and the referee for their comments on earlier drafts of this paper.

References

BRON, C. and KERBOSCH, J. (1973). Algorithm 457: Finding All Cliques of an Undirected Graph, *CACM*, Vol. 16 No. 9, pp. 575-577.

CHASE, S. M. (1970). Analysis of Algorithms for Finding All Spanning Trees of a Graph, Report No. 401, Department of Computer Science, University of Illinois.

CHRISTOFIDES, N. (1975). *Graph Theory—An Algorithmic Approach*, Academic Press, London.

CRESPI-REGHIZZI, S. and MORPURGO, R. (1968). A Graph Theory Oriented Extension to Algol, *Calcolo*, Vol. 5 No. 4, pp. 1-43.

DEO, N. (1974). *Graph Theory with Applications to Engineering and Computer Science*, Prentice-Hall.

EVEN, S. (1973). *Algorithmic Combinatorics*, Macmillan, New York.

FRIEDMAN, D. (1968). GRASPE Graph Processing: A LISP Extension, Report TNN-84, Computation Center, University of Texas, Austin.

GARSIDE, G. R. and PINTELAS, P. E. (1978). *GRAAP Reference Manual*, Computing Laboratory, University of Bradford.

GERHARDS, L. and LINDENBERG, W. (1979). Clique Detection for Nondirected Graphs: Two New Algorithms, *Computing*, Vol. 21, pp. 295-322.

JOHNSTON, H. C. (1976). Cliques of a Graph—Variations on the Bron-Kerbosch Algorithm, *Internat. J. Computing Information Sci.*, Vol. 5 No. 3, pp. 209-238.

KING, G. A. (1970). *A Graph-theoretic Programming Language*, Ph.D. thesis, University of West Indies.

PAULL, M. C. and UNGER, S. H. (1959). Minimizing the Number of States in Incompletely Specified Sequential Switching Functions, *IRE Trans. Electronic Computers, EC-8*, pp. 356-367.

PINTELAS, P. E. (1976). *An Algol 68-R Package for Handling Sets and Graphs*, Ph.D. thesis, University of Bradford.

RHEINBOLT, W. C., BASILI, V. R. and MESZTENYI, C. K. (1972). On a Programming Language for Graph Algorithms, *BIT*, Vol. 12, pp. 220-241.

TARJAN, R. (1972). Depth-first Search and Linear Graph Algorithms, *SIAM J. Comput.*, Vol. 1, pp. 146-160.

Book reviews

Automatic Speech and Speaker Recognition, edited by N. Rex Dixon and Thomas B. Martin, 1978; 428 pages. (*IEEE Press*)

This book is a collection of 38 papers on various aspects of speech and speaker recognition, originally published between 1972 and 1978. For someone wishing to get an indepth view of the field it would certainly save a lot of work in the library. The papers range widely and there are contributions from the UK, Japan and the Netherlands as well as from the United States.

The reader will find he needs to have a wide ranging preparation if he is to read and understand everything (acoustics, anatomy, phonetics, linguistics, mathematics, computing) but this is typical of the field. Five review papers on speech recognition and two on speaker recognition will provide a good overview of the main developments in the field for the non-specialist. If one believes, as does the reviewer, that progress in this area is crucial to a more effective use of computers in our society as a whole, the book contains evidence of solid if not dramatic progress in the last decade.

P. G. RAYMONT (Manchester)

Computer Security, by D. K. Hsiao, D. S. Kerr and S. E. Madnick, 1979; 299 pages. (*Academic Press*, $18·00)

This is another of a series of inter-related American research publications on computer security, and nearly 50 percent of it consists of references to other similar works. For once, however, the topic is seen as a wider ranging problem than teleprocessing access control, although this is still the major issue considered, whilst other important areas, such as power supply, flooding, corrupt input information and incorrectly timed use of file information, get but passing mention.

Some parts of the book are encouragingly written in layman's language, whilst others go into some depth on software matters. However technical the cause of problems, they have to be appreciated and evaluated by ordinary managers so that effective action may be taken. This volume provides a helpful survey of many aspects of computer security, but adds little new to the available information on the topic.

A. J. THOMAS (Sunbury on Thames)

Downloaded from https://academic.oup.com/comjnl/article/23/3/237/375193 by guest on 09 April 2024