Algorithm 457

Finding All Cliques of an Undirected Graph [H]

Coen Bron* and Joep Kerbosch† [Recd. 27 April 1971 and 23 August 1971]

* Department of Mathematics † Department of Industrial Engineering, Technological University Eindhoven, P.O. Box 513, Eindhoven, The Netherlands

Present address of C. Bron: Department of Electrical Engineering, Twente University of Technology, P.O. Box 217, Enschade, The Netherlands.

Key Words and Phrases: cliques, maximal complete subgraphs, clusters, backtracking algorithm, branch and bound technique, recursion

CR Categories: 3.71, 5.32

Language: Algol

Description

Introduction. A maximal complete subgraph (clique) is a complete subgraph that is not contained in any other complete subgraph.

A recent paper [1] describes a number of techniques to find maximal complete subgraphs of a given undirected graph. In this paper, we present two backtracking algorithms, using a branch-and-bound technique [4] to cut off branches that cannot lead to a clique.

The first version is a straightforward implementation of the basic algorithm. It is mainly presented to illustrate the method used. This version generates cliques in alphabetic (lexicographic) order.

The second version is derived from the first and generates cliques in a rather unpredictable order in an attempt to minimize the number of branches to be traversed. This version tends to produce the larger cliques first and to generate sequentially cliques having a large common intersection. The detailed algorithm for version 2 is presented here.

Description of the algorithm—Version 1. Three sets play an important role in the algorithm. (1) The set *compsub* is the set to be extended by a new point or shrunk by one point on traveling along a branch of the backtracking tree. The points that are eligible to extend *compsub*, i.e. that are connected to all points in *compsub*, are collected recursively in the remaining two sets. (2) The set *candidates* is the set of all points that will in due time serve as an extension to the present configuration of *compsub*. (3) The set *not* is the set of all points that have at an earlier stage already served as an extension of the present configuration of *compsub* and are now explicitly excluded. The reason for maintaining this set *not* will soon be made clear.

The core of the algorithm consists of a recursively defined extension operator that will be applied to the three sets just described. It has the duty to generate all extensions of the given configuration of *compsub* that it can make with the given set of candidates and that do not contain any of the points in *not*. To put it differently: all extensions of *compsub* containing any point in *not* have already been generated. The basic mechanism now consists of the following five steps:

Step 1. Selection of a candidate.
Step 2. Adding the selected candidate to *compsub*.
Step 3. Creating new sets *candidates* and *not* from the old sets by removing all points not connected to the selected candidate (to remain consistent with the definition), keeping the old sets in tact.
Step 4. Calling the extension operator to operate on the sets just formed.
Step 5. Upon return, removal of the selected candidate from *compsub* and its addition to the old set *not*.

We will now motivate the extra labor involved in maintaining the sets *not*. A necessary condition for having created a clique is that the set *candidates* be empty; otherwise *compsub* could still be extended. This condition, however, is not sufficient, because if now *not* is nonempty, we know from the definition of *not* that the present configuration of *compsub* has already been contained in another configuration and is therefore not maximal. We may now state that *compsub* is a clique as soon as both *not* and *candidates* are empty.

If at some stage *not* contains a point connected to all points in *candidates*, we can predict that further extensions (further selection of candidates) will never lead to the removal (in Step 3) of that particular point from subsequent configurations of *not* and, therefore, not to a clique. This is the branch and bound method which enables us to detect in an early stage branches of the backtracking tree that do not lead to successful endpoints.

A few more remarks about the implementation of the algorithm seem in place. The set *compsub* behaves like a stack and can be maintained and updated in the form of a global array. The sets *candidates* and *not* are handed to the extensions operator as a parameter. The operator then declares a local array, in which the new sets are built up, that will be handed to the inner call. Both sets are stored in a single one-dimensional array with the following layout:

| *not* | *candidates*

index values: 1.....*ne*.....*ce*....

The following properties obviously hold:

1. $ne \leq ce$
2. $ne = ce$:empty (*candidates*)
3. $ne = 0$:empty (*not*)
4. $ce = 0$:empty (*not*) and empty (*candidates*)
   = clique found

If the selected candidate is in array position $ne + 1$, then the second part of Step 5 is implemented as $ne := ne + 1$.

In version 1 we use element $ne + 1$ as selected candidate. This strategy never gives rise to internal shuffling, and thus all cliques are generated in a lexicographic ordering according to the initial ordering of the candidates (all points) in the outer call.

For an implementation of version 1 we refer to [3].

Description of the algorithm—Version 2. This version does not select the candidate in position $ne + 1$, but a well-chosen candidate from position, say s. In order to be able to complete Step 5 as simply as described above, elements s and $ne + 1$ will be interchanged as soon as selection has taken place. This interchange does not affect the set *candidates* since there is not implicit ordering.

Fig. 1. Random graphs show the computing time per clique (in ms) versus dimension of the graph (in brackets: total number of cliques in the test sample).

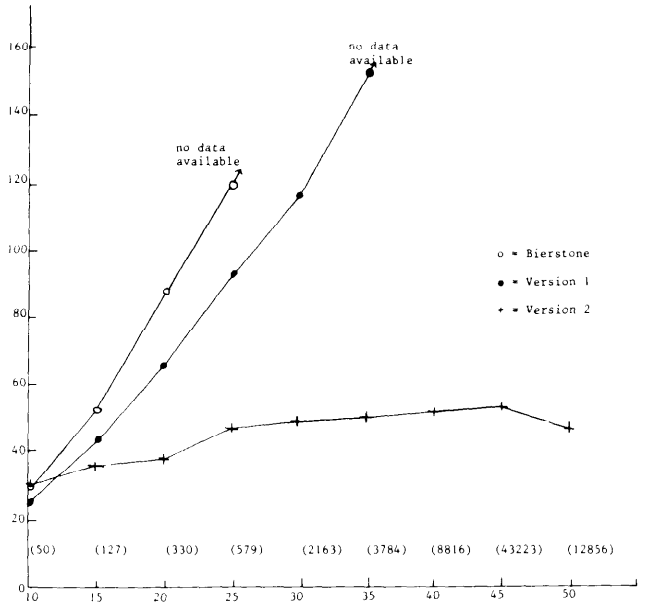
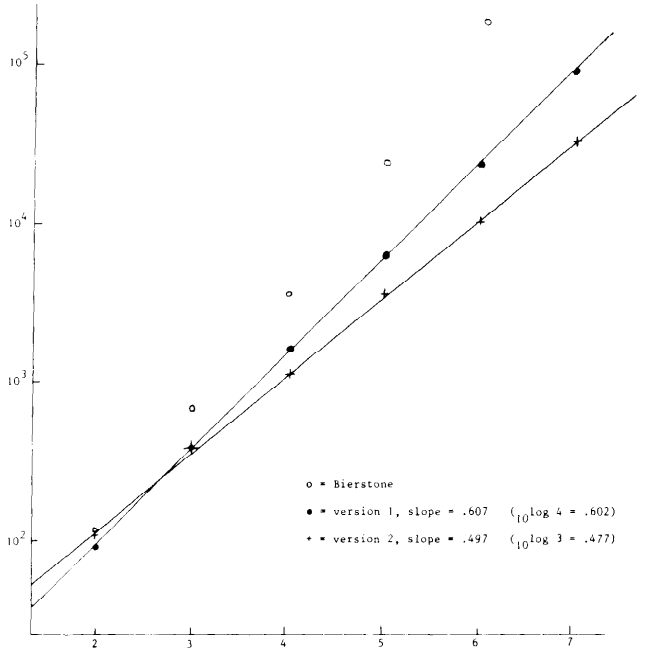


Fig. 2. Moon-Moser graphs show the computing time (in ms) versus k. Dimension of the graph = $3k$. Plotted on logarithmic scale.



The selection does affect, however, the order in which the cliques are eventually generated.

Now what do we mean by "well chosen"? The object we have in mind is to minimize the number of repetitions of Steps 1–5 inside the extension operator. The repetitions terminate as soon as the bound condition is reached. We recall that this condition is formulated as: there exists a point in *not* connected to all points in *candidates*. We would like the existence of such a point to come about at the earliest possible stage.

Let us assume that with every point in *not* is associated a counter, counting the number of candidates that this point is not connected to (*n*umber *o*f *d*isconnections). Moving a selected candidate into *not* (this occurs after extension) decreases by one all counters of the points in *not* to which it is disconnected and introduces a new counter of its own. Note that no counter is ever decreased by more than one at any one instant. Whenever a counter goes to zero the bound condition has been reached.

Now let us fix one particular point in *not*. If we keep selecting candidates disconnected to this fixed point, the counter of the fixed point will be decreased by one at every repetition. No other counter can go down more rapidly. If, to begin with, the fixed point has the lowest counter, no other counter can reach zero sooner, as long as the counters for points newly added to *not* cannot be smaller. We see to this requirement upon entry into the extension operator, where the fixed point is taken either from *not* or from the original *candidates*, whichever point yields the lowest counter value after the first addition to *not*. From that moment on we only keep track of this one counter, decreasing it for every next selection, since we will only select disconnected points.

The Algol 60 implementation of this version is given below.

Discussion of comparative tests. Augustson and Minker [1] have evaluated a number of clique finding techniques and report an algorithm by Bierstone [2] as being the most efficient one.

[1] Bierstone's algorithm as reported in [1] contained an error. In our implementation the error was corrected. The error was independently found by Mulligan and Corneil at the University of Toronto, and reported in [6].

In order to evaluate the performance of the new algorithms, we implemented the Bierstone algorithm[1] and ran the three algorithms on two rather different testcases under the Algol system for the EL-X8.

For our first testcase we considered random graphs ranging in dimension from 10 to 50 nodes. For each dimension we generated a collection of graphs where the percentage of edges took on the following values: 10, 30, 50, 70, 90, 95. The cpu time per clique for each dimension was averaged over such a collection. The results are graphically represented in Figure 1.

The detailed figures [3] showed the Bierstone algorithm to be of slight advantage in the case of small graphs containing a small number of relatively large cliques. The most striking feature, however, appears to be that the time/clique for version 2 is hardly dependent on the size of the graph.

The difference between version 1 and "Bierstone" is not so striking and may be due to the particular Algol implementation. It should be borne in mind that the sets of nodes as they appear in the Bierstone algorithm were coded as one-word binary vectors, and that a sudden increase in processing time will take place when the input graph is too large for "one-word representation" of its subgraphs.

The second testcase was suggested by the referee and consisted of regular graphs of dimensions $3 \times k$. These graphs are constructed as the complement of k disjoint 3-cliques. Such graphs contain 3^k cliques and are proved by Moon and Moser [5] to contain the largest number of cliques per node.

In Figure 2 a logarithmic plot of computing time versus k is presented. We see that both version 1 and version 2 perform significantly better than Bierstone's algorithm. The processing time for version 1 is proportional to 4^k, and for version 2 it is proportional to $(3.14)^k$ where 3^k is the theoretical limit.

Another aspect to be taken into account when comparing algorithms is their storage requirements. The new algorithms presented in this paper will need at most $\frac{1}{2}M(M+3)$ storage locations to contain arrays of (small) integers where M is the size of largest connected component in the input graph. In practice this limit will only be approached if the input graph is an almost com-

plete graph. The Bierstone algorithm requires a rather unpredictable amount of store, dependent on the number of cliques that will be generated. This number may be quite large, even for moderate dimensions, as the Moon-Moser graphs show.

Finally it should be pointed out that Bierstone's algorithm does not report isolated points as cliques, whereas the new algorithm does. Either algorithm can, however, be modified to produce results equivalent to the other. Suppression of 1-cliques in the new algorithm is the simplest adaption.

Acknowledgments. The authors are indebted to H.J. Schell for preparation of the test programs and collection of performance statistics. Acknowledgments are also due to the referees for their valuable suggestions.

References

1. Augustson, J.G., and Minker, J. An analysis of some graph theoretical cluster techniques, *J. ACM 17* (1970), 571–588.
2. Bierstone, E. Unpublished report. U of Toronto.
3. Bron, C., Kerbosch, J.A.G.M., and Schell, H.J. Finding cliques in an undirected graph. Tech. Rep. Technological U. of Eindhoven, The Netherlands.
4. Little, John D.C., et al. An algorithm for the traveling salesman problem. *Oper. Res. 11* (1963), 972–989.
5. Moon, J.W., and Moser, L. On cliques in graphs. *Israel J. Math.* 3 (1965), 23–28.
6. Mulligan, G.D., and Corneil, D.G. Corrections to Bierstone's algorithm for generating cliques. *J. ACM 19* (Apr. 1972), 244–247.

Algorithm

```
procedure output maximal complete subgraphs 2(connected, N);
  value N; integer N;
  Boolean array connected;
comment The input graph is expected in the form of a symmetrical
  Boolean matrix connected. N is the number of nodes in the
  graph. The values of the diagonal elements should be true;
begin
  integer array ALL, compsub[1 : N];
  integer c;
  procedure extend version 2(old, ne, ce);
    value ne, ce; integer ne, ce;
    integer array old;
  begin
    integer array new[1 : ce];
    integer nod, fixp;
    integer newne, newce, i, j, count, pos, p, s, sel, minnod;
    comment The latter set of integers is local in scope but need
      not be declared recursively;
    minnod := ce; i := nod := 0;
DETERMINE EACH COUNTER VALUE AND LOOK FOR MINIMUM:
    for i := i + 1 while i ≤ ce ∧ minnod ≠ 0 do
    begin
      p := old[i]; count := 0; j := ne;
COUNT DISCONNECTIONS:
      for j := j + 1 while j ≤ ce ∧ count < minnod do
        if ¬ connected[p, old[j]] then
        begin
          count := count + 1;
SAVE POSITION OF POTENTIAL CANDIDATE:
          pos := j
        end;
TEST NEW MINIMUM:
      if count < minnod then
      begin
        fixp := p; minnod := count;
        if i ≤ ne then s := pos
        else
        begin s := i; PREINCR: nod := 1 end
      end NEW MINIMUM;
    end i;
    comment If fixed point initially chosen from candidates then
      number of disconnections will be preincreased by one;
BACKTRACKCYCLE:
    for nod := minnod + nod step −1 until 1 do
    begin
INTERCHANGE:
      p := old[s]; old[s] := old[ne + 1];
      sel := old[ne + 1] := p;
FILL NEW SET not:
      newne := i := 0;
      for i := i + 1 while i ≤ ne do
        if connected[sel, old[i]] then
        begin newne := newne + 1; new[newne] := old[i] end;
FILL NEW SET cand:
      newce := newne; i := ne + 1;
      for i := i + 1 while i ≤ ce do
        if connected[sel, old[i]] then
        begin newce := newce + 1; new[newce] := old[i] end;
ADD TO compsub:
      c := c + 1; compsub[c] := sel;
      if newce = 0 then
      begin
        integer loc;
        outstring(1, 'clique = ');
        for loc := 1 step 1 until c do
          outinteger(1, compsub[loc])
      end output of clique
      else
      if newne < newce then extend version 2(new, newne, newce);
REMOVE FROM compsub:
      c := c − 1;
ADD TO not:
      ne := ne + 1;
      if nod > 1 then
      begin
SELECT A CANDIDATE DISCONNECTED TO THE FIXED POINT:
        s := ne;
LOOK: FOR CANDIDATE:
        s := s + 1;
        if connected[fixp, old[s]] then go to LOOK
      end selection
    end BACKTRACKCYCLE
  end extend version 2;
  for c := 1 step 1 until N do ALL[c] := c;
  c := 0; extend version 2(ALL, 0, N)
end output maximal complete subgraphs 2;
```