Finding cliques in an undirected graph

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

FINDING CLIQUES IN A UNDIRECTED GRAPH

door

Bron, C.   Department of Mathematics

                    Group: Fundamental Programming

Kerbosch, J.A.G.M. Department of Industrial Engineering

Schell, H.J.   Group: Operations Research

Contents   Page

1. Introduction.

In 1970 Poeth [7] , a student at the department of Industrial Engineering
asked us to develop an algorithm for finding all cliques in an undirected
graph. He needed the results for analysing the data of a sociological
survey in an organisation. These data resulted in a symmetrical incidence-
matrix, i.e.: if in a test the mutual appreciation of person i and person
j was significant then the element (i, j) was set to "1" otherwise to "0".
The cliques were defined as non-extendable groups such that each pair of
persons within the group had a good relationship. The problem of finding
cliques is the same as finding maximum complete subgraphs in a graph.
The same sociological method was used by Schaay [8] . In a computer-
program he listed all complete subgraphs of 3 vertices and constructed
the maximal complete subgraphs from this list by hand, a tedious task.
At the department of Industrial Engineering, a back-tracking algorithm
was developed, based on the concepts, developed in [4] . Later on, this
algorithm was improved considerably in collaboration with the group Fundamental
Programming of the Department of Mathematics, leading to two algorithms.

2. Algorithms.

The following algorithms generate all maximal complete subgraphs of a given
undirected graph. A maximal complete subgraph is a complete subgraph that
cannot be extended with another point of the mother graph and yet remain
complete. In the following we will term such a maximal complete subgraph
a clique.
Both algorithms are in essence backtracking algorithms, using a "branch and
bound" technique [4] to cut off branches that cannot lead to a clique. The
first version, to be described directly below, is a straightforward
implementation of the algorithm, and generates all cliques in alphabetic
(lexicographic) order. The second version is based on the first and generates
cliques in a rather unpredictable order in an attempt to minimize the number
of branches to be traversed. Performance tests on graphs containing a large
number of cliques have shown the second version to be far superoir (a
performance factor of up to 5 has been encountered). Nevertheless it is
felt that both versions should be presented in order to explain the algorithm.

Three sets play an important role in the algorithm, viz.
1) the set "compsub": the set to be extended by a new point or shrunk by one
   point on travelling along the branches 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 two sets, viz.
2) the set "candidates": the set of all points that will in due time serve
   as an extension to the present configuration of "compsub".
3) the set "not": the set of all points that have at an earlier stage already
   served as an extension to the present configuration of "compsub" and are
   now explicitly excluded. The reason for maintaining the 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 <u>not</u> 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 5 steps:
1: selection of a candidate
2: adding the selected candidate to "compsub"
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), thereby keeping the old sets in tact
4: calling the extension operator
5: upon return the selected condidate is removed from "compsub" and moved
   into the old set "not".

We will now motivate the extra labour involved in recursively 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 at this stage the set "not" is non-empty, we
know from the definition of "not" that the present configuration of "compsub"
has at an earlier stage been contained in another configuration and is
therefore not maximal. We may now state that "comsub" is a clique as soon as
both "not" and "candidates" are empty.

If at some stage we have a set "not" containing 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". Such extensions will not lead to an empty set "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 extension 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:

$$\text{"not"} \qquad \text{"candidates"}$$

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

From this the following properties may be derived:

1) $ne \leq ce$

2) $ne = ce$: "candidates" empty

3) $ne = 0$: "not" empty

4) $ce = 0$: "not" empty, "candidates" empty $\rightarrow$ clique found

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

Using "$ne + 1$" as selected candidate never gives rise to internal shuffling
and therefore all cliques are generated in a lexicographic ordering
according to the initial ordering of the candidates (all points) in
the outer call.

We will first present the main procedure, and then seperately, the text
of the extension operator for both versions of the algorithm.

```
procedure output maximal complete subgraphs(connected, N); value N;
integer N; comment number of points in graph;
boolean array connected; comment symmetrical matrix;
begin integer array ALL, compsub[1 : N]; integer c;
        procedure extend(old, ne, ce); value ne, ce; integer ne, ce;
        integer array old;
         "BODY OF extend";
        for c:=1 step 1 until N do ALL[c]:=c; c:=0;
        comment initially all points are connected to the points in the empty set
                "compsub" and hence candidates;
        extend(ALL, 0, N)
end output maximal complete subgraphs;


procedure extend(old, ne, ce); value ne, ce; integer ne, ce; integer array old;
BODY OF extend VERSION 1 :
begin integer array new[1 : ce];
        comment in the worst case the new sets will be almost as large as the old ones;
        boolean allcon, sucexp; integer i, j, p, sel, newne, newce;
        repeat TEST IF sucCES MAY BE expECTED FROM FURTHER EXTENSION:
                sucexp:=true; i:=0;
SCAN not:       while sucexp and (i:=i + 1) ≤ ne do
                begin p:=old[i]; allcon:=true; j:=ne;
SCAN candidates:    while allcon and (j:=j + 1) ≤ ce do allcon:=connected[p, old[j]];
                    sucexp:= not allcon
                end;
                if sucexp do
                begin SELECT CANDIDATE: sel:=old[ne + 1];
                newne:= i:= 0;
FILL NEW SET not:   while (i:=i + 1) ≤ ne do
                    if connected[sel, (p:=old[i])] do new[(newne:=newne + 1)]:=p;
                    newce:=newne; comment selected candidate is skipped;
FILL NEW SET cand:  while (i:=i + 1) ≤ ce do
                    if connected[sel, (p:=old[i])] do new[(newce:=newce + 1)]:=p;
ADD TO compsub:     compsub[(c:=c + 1)]:= sel;
                    if newce = 0 then OUTPUT VALUES OF compsub 1 THROUGH c
                                 else if newne < newce do extend(new, newne, newce);
REMOVE FROM compsub: c:= c - 1;
ADD TO not:          ne:= ne + 1;
                end
        until not sucexp
end extend VERSION 1;
```

The second version of the algorithm 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 no implicit ordering. 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 through 5 inside the extension operator. The repetitions terminate as soon as the "bound condition" is reached. We recall that the bound 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 points in "candidates" to which this point is not connected (the number of disconnections). 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 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 be decreased 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 the above requirement upon entry 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.

We will now present the optimized version of the extension operator (see next page):

```
procedure extend(old, ne, ce); value ne, ce; integer ne, ce; integer array old;
BODY OF extend VERSION 2 :
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:=1; nod:=0;
SET INITIAL nod VALUE:
         repeat DETERMINE EACH COUNTER VALUE AND LOOK FOR MINIMUM:
                p:=old[i]; count:=0; j:=ne;
COUNT DISCONNECTIONS: repeat j:=j + 1;
                      if not connected[p, old[j]] do
                      begin count:=count + 1;
                            SAVE POSITION OF POTENTIAL CANDIDATE: pos:=j
                      end
                  until j=ce;
TEST NEW MINIMUM:     if count < minnod do
                     begin fixp:=p; minnod:=count;
                         if i ≤ ne then s:=pos
                                   else begin s:=i; PREINCREASE: nod:=1 end
                     end;
                     i:= i + 1
         until i > ce or minnod = 0;
         nod:=minnod + nod;
         comment possible pre-incease by one if fixed point initially in "candidates";
         while nod > 0 do
         begin INTERCHANGE: p:=old[s]; old[s]:=old[ne+1]; sel:=old[ne+1]:=p;
               newne:=i:=0;
FILL NEW SET not:    while (i:=i + 1) ≤ ne do
                     if connected[sel, (p:=old[i])] do new[(newne:=newne + 1)]:=p;
               newce:=newne;
               comment next i-increase will skip over selected candidate;
FILL NEW SET cand:   while (i:=i + 1) ≤ ce do
                     if connected[sel, (p:=old[i])] do new[(newce:=newce + 1)]:=p;
ADD TO compsub:      compsub[(c:=c + 1)]:=sel;
               if newce = 0 then OUTPUT VALUES OF compsub 1 THROUGH c
                            else if newne < newce do extend(new, newne, newce);
REMOVE FROM compsub: c:=c - 1; ADD TO not: ne:=ne + 1;
               if (nod:=nod - 1) > 0 do
               begin SELECT A CANDIDATE DISCONNECTED TO THE FIXED POINT:
                     s:=ne; repeat s:=s + 1 until not connected[fixp, old[s]]
               end
         end
     end
```

3. Discussion of comperative 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.
In order to evaluate the performance of the new algorithms we implemented
the Bierstone algorithm[*) 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 fig. 1. This averaging
hides the influence of the percentage of edges (p); the tendency, shown in
fig. 1 becomes more significant with increasing p. (See Appendix).
The detailed figures (appendix) showed the Bierstone algorithm to be of slight
advantage in the case of small graphs containing a small number of relative-
ly 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 of C.A.C.M. and consisted of
graphs of dimension 3 $*$ 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 [6] to contain the largest number of cliques per node.

Footnote:[*) 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
a paper submitted to J.A.C.M..

[+) For our definition of random graphs see the Appendix.

In Fig. 2 a logarithmic plot of computing time vs. 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. Detailed computional results are given in the Appendix. Our results were confirmed by Mulligan [6].

Another aspect to be taken into account when comparing algorithms is their storage requirement. 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 complete 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.

Random graphs. Computing time per clique (in ms) versus dimension of the graph. In brackets: total number of cliques in the test sample.

fig. 1
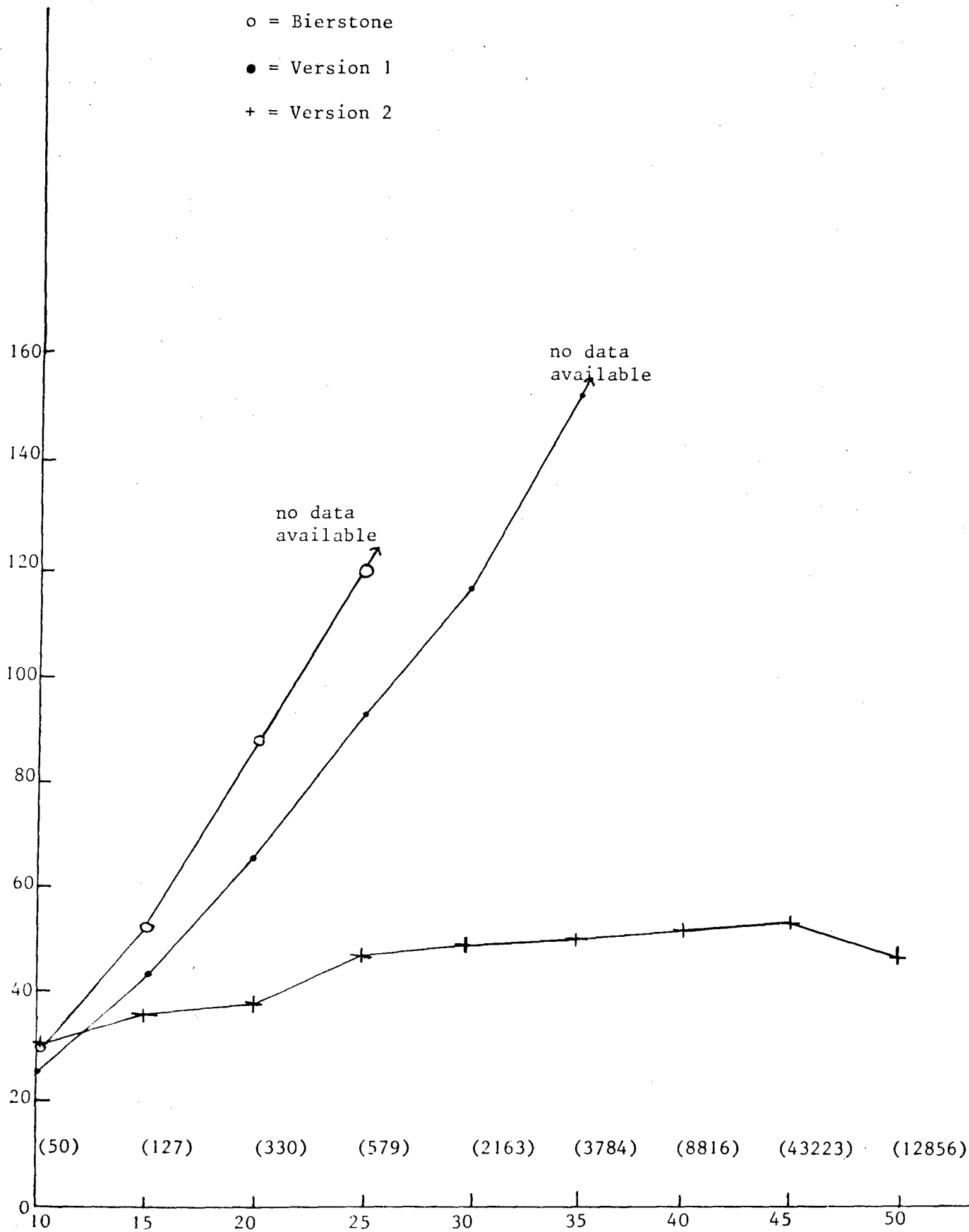
Moon-Moser graphs. Computing time in ms versus k. Dimension of the graph $3k$. Plotted on logarithmic scale.
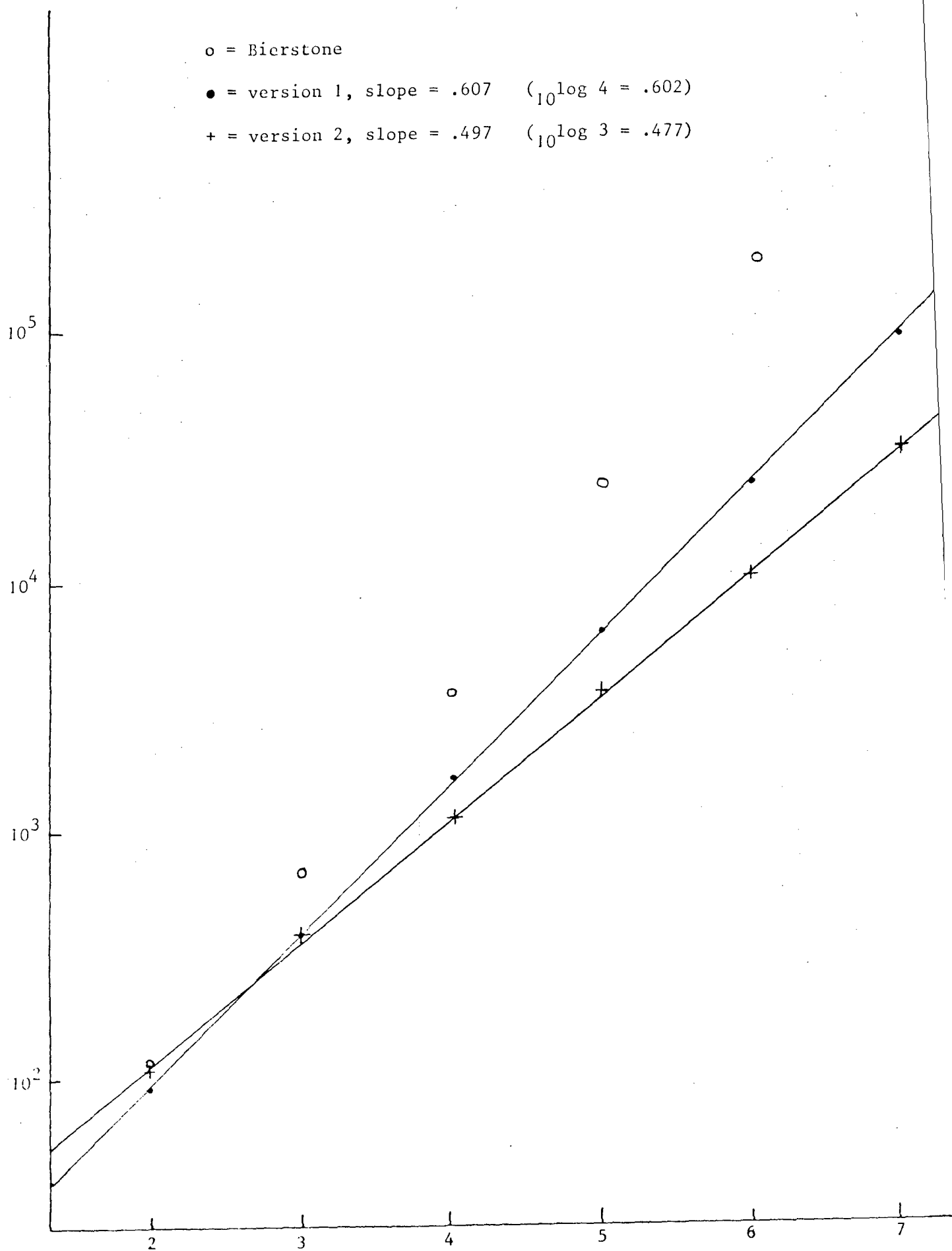
Fig. 2

o = Bierstone

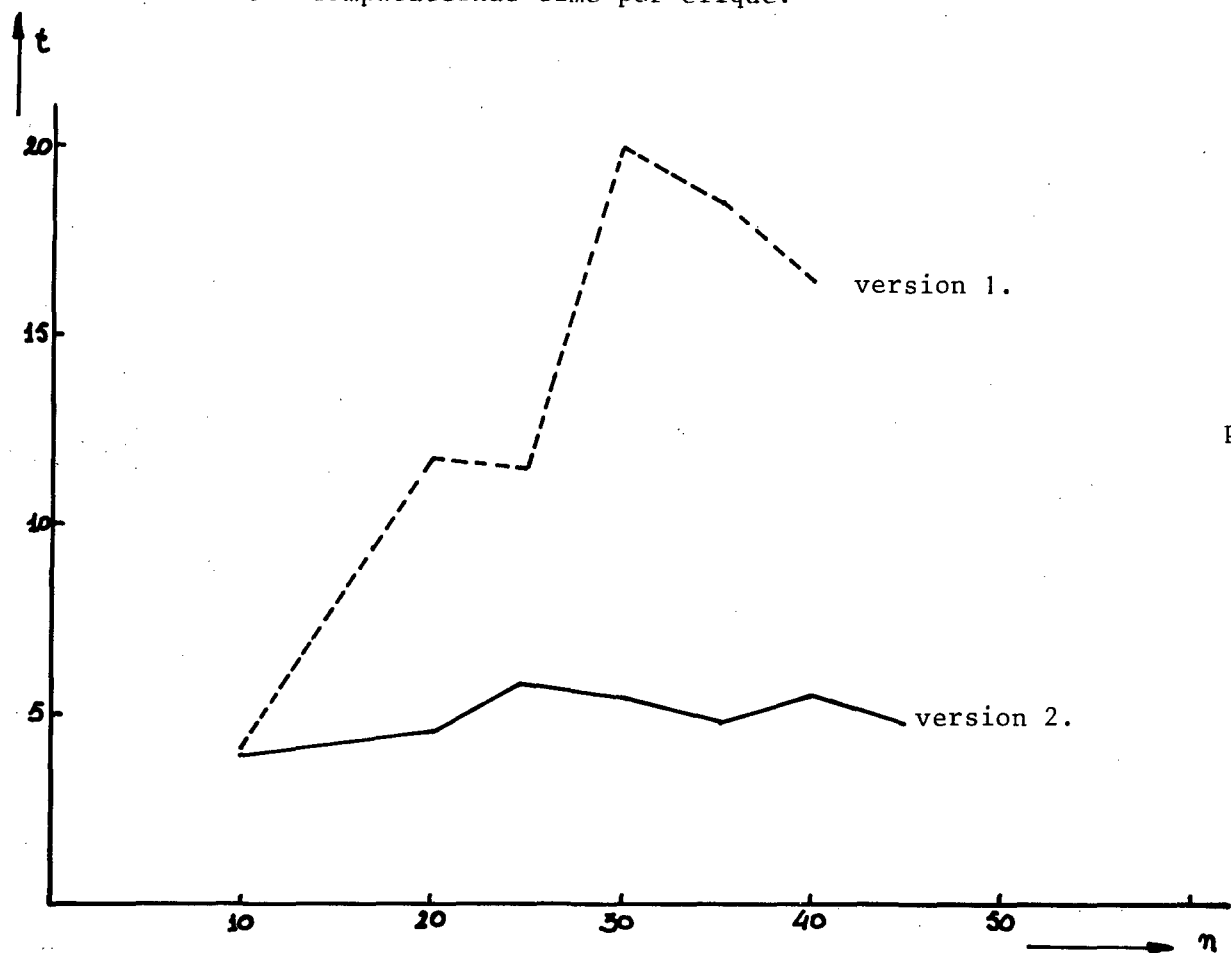● = version 1, slope = .607 $(_{10}\log 4 = .602)$

+ = version 2, slope = .497 $(_{10}\log 3 = .477)$

Fig. 3. The effect of p.

t = computational time per clique.
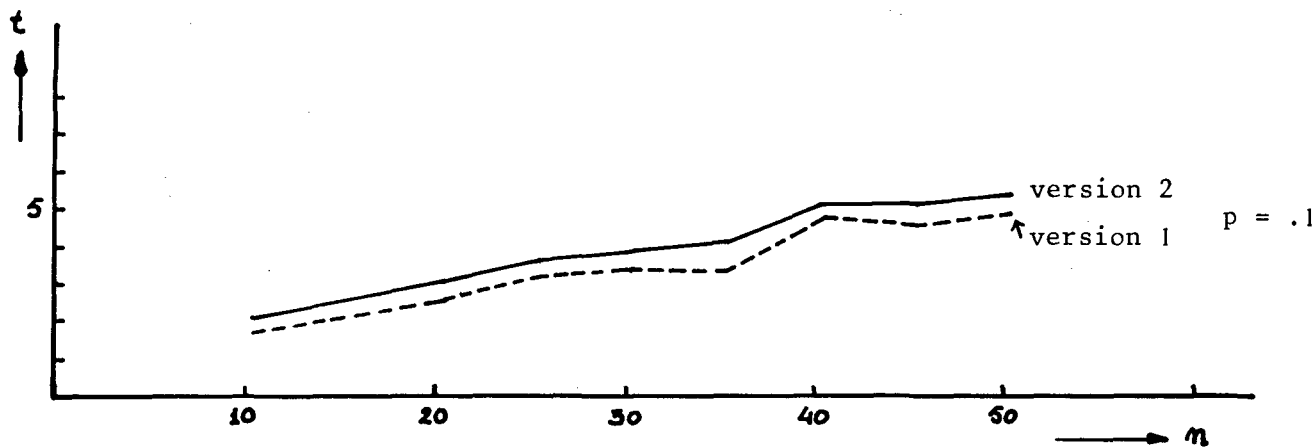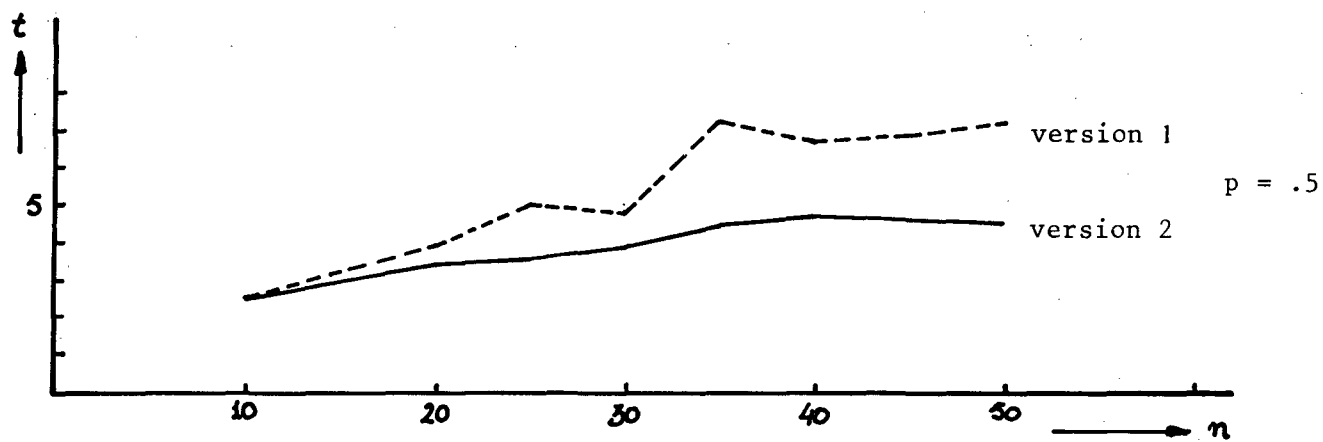
version 1.

p = .9

version 2.

version 1

p = .5

version 2

version 2

p = .1

version 1

4. Acknowledgements.

5. References:

[1] Augustson,J.G., and Minker, J., An analysis of some graphtheoretical cluster techniques, Journal A.C.M. 17 (1970), 571-588.

[2] Bierstone, E., Unpublished report, Univ. of Toronto.

[3] Bron, Coen and Kerbosch, Joep, A.G.M., Finding all cliques in an un-directed Graph, Comm. A.C.M., to appear.

[4] Little,John, D.C., et al. An Algorithm for the traveling salesman problem, J. Oper. Res. 11 (1963), 972-989.

[5] Moon, J.W. and Moser, L.,Isr. J. of Math., 3 (1965), 23-28.

[6] Mulligan, Gordon D., Algorithms for finding Cliques of a Graph, Oct. 1971, Master's Thesis, Department of Computer Science, Univ. of Toronto.

[7] Poeth, G.J.M., Organisaties en effektiviteit, June 1970, Master's Thesis, Department of Industrial Engineering, Univ. of Technology, Eindhoven.

[8] Schaay, J.A.L., Organisatie, formeel en feitelijk, Doctor's Thesis, 1969, Department of Social Sciences, Univ. of Utrecht, page 123-125.

6. Appendix.

6.1. Random Graphs.

In order to generate a random graph of dimension n and percentage of edges p, we associate with each pair of numbers $\{i,j\}$, $1 \le i < j \le n$, a probability p. With this probability, we draw at random whether the edge (i,j) exists or not.

This implies, the the actual fraction of edges in the graph, $\hat{p}$, may be slightly different from p. Some part of the variance of our results may be due to this (rather unusual) method of generating random graphs. Yet it does not disturb our conclusions. (SeeMulligan [6]).

6.2. Computational time per clique.

As an extension of figure 1 we present the following tables. The figures are average of two samples. The variance of the computational time per clique was small. Figure 3 illustrates the effect of p.

Computational time per clique $\times$ 100 seconds.

Version 1:

		10	20	25	$\underset{=}{n}$ 30	35	40	45	50
	.1	1.6	2.5	3.1	3.3	3.2	4.6	4.4	4.7
	.3	1.8	3.1	3.0	3.8	3.7	5.0	4.9	6.6
p .5		2.3	3.8	4.8	4.8	7.0	6.5	6.7	7.0
= .7		2.4	4.7	4.3	9.0	9.4	8.9	14.8	10.5
	.9	4.2	11.9	11.7	20.0	18.7	16.6	-	-

Version 2:

		10	20	25	$\underset{=}{n}$ 30	35	40	45	50
	.1	2.0	3.0	3.5	3.8	4.0	5.0	5.0	5.8
	.3	2.2	3.0	3.5	3.6	3.6	4.5	4.7	6.3
p .5		2.3	3.2	3.3	3.8	4.2	4.5	4.4	4.3
= .7		2.7	4.0	2.7	4.3	4.4	4.6	5.0	4.5
	.9	4.0	4.6	5.9	5.5	4.9	5.5	4.9	-

6.3. Total computational time.

The following tables give the average total computational time (on the EL-X8 computer), to find all cliques. For fixed values of n and p, the number of cliques in a graph has a large variance; therefore, these tables are only presented to indicate the order of magnitude.

Computational time to find all cliques in seconds.

Version 1:

					$\underset{=}{n}$				
		10	20	25	30	35	40	45	50
	.1	.1	.6	.8	1.2	1.8	2.7	3.4	4.3
	.3	.2	1.1	1.7	3.1	4.0	8	9	13
p	.5	.2	1.8	4.4	8.0	17	30	42	70
=	.7	.3	4.1	13.0	35	80	150	485	540
	.9	.4	9	43	75	480	1030	–	–

Version 2:

					$\underset{=}{n}$				
		10	20	25	30	35	40	45	50
	.1	.2	.7	.9	1.5	2.1	2.9	3.9	4.6
	.3	.2	1.0	1.9	2.9	3.9	7.2	8.8	12
p	.5	.2	1.6	3.0	6.3	10.0	20	28	45
=	.7	.3	3.2	8.3	16.5	37.0	80	163	230
	.9	.4	3.5	21.5	21.0	125.7	380	1940	–