Finding the maximal strong components in a directed graph.

Given a directed graph, i.e. a set of vertices and a set of directed edges, each leading from one vertex to another, it is requested to partition the vertices into so-called "maximal strong components". A strong component is a set of vertices, such that the edges between them provide a directed path from any vertex of the set to any vertex of the set and vice versa; a single vertex is a special case of a strong component: then the path can be empty. A maximal strong component is a strong component to which no further vertices can be added.

In order to establish this partitioning, we have to be able to make two kinds of assertions: the assertion that vertices belong to the same strong component, but also --because we have to find maximal strong components-- the assertion that vertices do not belong to the same strong component.

For the first type of assertion, we may use the following

Theorem 1. Cyclically connected vertices belong to the same strong component.

Besides (directed) connections between individual vertices, we can talk about directed connections between different strong components: we say that there is a connection from a strong component A to a strong component B if there exists a directed edge from a vertex of A to a vertex of B. Because A and B are strong components, there is then a path from any vertex of A to any vertex of B. And as a result, Theorem 1 can be generalized into

Theorem 1a. Vertices of cyclically connected strong components belong to the same strong component.

Corollary 1. A non-empty graph has at least one maximal strong component without outgoing edges.

So much for theorems asserting that vertices belong to the same strong component. Because for different points to be in the same strong component, there must be paths between them in both ways, assertions that vertices do not belong to the same strong component can be made on account of

Theorem 2. If the vertices are subdivided into two sets svA and svB, such that there exist no edges originating in a vertex of svA and terminating in a vertex of svB, then

firstly: the set of maximal strong components does not depend on the presence or absence of edges originating in a vertex of svB and terminating in a vertex of svA, and

secondly: no strong component comprises vertices from both sets.

From Theorem 2 it follows that as soon as a strong component without outgoing edges has been found, we can take its vertices as set svA and conclude that this strong component is a maximal strong component and that all ingoing edges of svA can further be ignored. We conclude

Theorem 2a. A strong component whose outgoing edges, if any, are all ingoing edges of maximal strong components, is itself a maximal strong component.

Or, to put it in another way: once the first maximal strong component without outgoing edges --the existence of which is guaranteed by Corollary 1-- has been found --identified as such by being a strong component without out-

going edges-- the remaining maximal strong components can be found by solving the problem for the graph consisting of the remaining vertices and only the given edges between them. Or, to put it in still another way, the maximal strong components of a graph can be ordered according to "age", such that each maximal strong component has only outgoing edges to "older" ones.

In order to be able to be a little bit more precise, we denote by

sv: the given set of vertices (a constant)

se: the given set of edges (a constant)

pv: a partitioning of the vertices of sv .

The final relation to be established can then be written as

R: $pv = MSC(se)$

in which for the fixed set sv the function MSC, i.e. the partitioning in Maximal Strong Components, is regarded as a function of the set of edges se.

The corresponding invariant relation is suggested by the standard technique of replacing a constant by a variable, se1 say, whose value will always be a subset of se:

P: $pv = MSC(se1)$.

Relation P is easily initialized for empty se1 --i.e. each vertex of sv is a maximal strong component all by itself-- . Because se1 is bounded in size by se , monotonically increasing se1 is guaranteed to terminate; if we can accomplish this under invariance of P , relation R has been established by the time that se1 = se . In our discussions it will be convenient also to have a name, se2 say, for the remaining edges, i.e. $se = se1 \mp se2$.

Our task is clearly to discover the most convenient order in which edges are to be added to se1 , where "convenience" is related to the ease with which the invariance of relation P is maintained. This, as we know, can also be phrased as: what is our general intermediate state, what types of pv-values do we admit? In order to describe such a general intermediate state, it seems practical to group the vertices of sv also in disjoint subsets (as we have done for the edges: se1 and se2). After all: we are interested in partitioning vertices!

The general intermediate state should be a generalization of both initial and final state. At the beginning, for none of the vertices it has been established to which maximal strong component in MSC(se) they belong, eventually it has been established for all vertices. Analogous to se1 we can introduce (initially empty and finally comprising all vertices) sv1, where

sv1 contains all vertices of sv , for which the maximal strong component in MSC(se) to which they belong has been identified.

We intend to use Theorem 2a for deciding that a strong component is a maximal one, that is, after having established something about <u>all</u> its outgoing edges. When we now identify:

se1 with the set of all processed edges, and

se2 with the set of all unprocessed edges, i.e. edges whose presence has not yet been taken into account,

then we see that

P1: all outgoing edges of vertices in sv1 are in se1.

It is, however, too crude to group all remaining vertices in a single set sv2. The way in which sv1 is defined implies that, each time a new maximal strong component of MSC(se) has been identified, all the vertices of that maximal strong component have to be transferred <u>together</u> to sv1. Between two such transfers, in general a number of edges have to be processed (i.e transferred from se2 to se1), and for the description of the intermediate states that have to be taken into account with respect to "processing one edge at a time", the remaining vertices have to be separated a little bit more subtly, viz, into two disjoint subsets, sv2 and sv3 say, (with sv = sv1 ∓ sv2 ∓ sv3), where sv3 contains the totally unprocessed vertices,

P2: no edge in se1 begins or ends at a vertex in sv3

(sv3 is initially equal to sv and finally empty).

Transfer from sv3 to sv1 can then take place in two steps: from sv3 to sv2 (one at a time) and from sv2 to sv1 (together with all other vertices from the same definite maximal strong component).

In other words: among the vertices of sv2 we shall try to build up (by enlarging se1) the next maximal strong component of MSC(se) to be transferred to sv1. The maximal strong components in MSC(se1) --note the argument!-- are such that they comprise either vertices from sv1 only, or vertices from sv2 only, or a (single) vertex from sv3. We propose a limitation on the connections that the edges of se1 provide between the maximal strong components in MSC(se1) that contain nodes from sv2 only: between those maximal strong components the edges of se1 shall provide no more and no less than a single directed path, leading from the "oldest" to the "youngest" one. We call these maximal strong components "the elements of the chain". This choice is suggested by the following considerations.

Firstly, we are looking for a cyclic path that would allow us to apply Theorem 1 or 1a , in order to decide that different vertices belong to the same maximal strong component. Under the assumption that we are free to prescribe which edge will be the next one to be added to se1 , there does not seem to be much advantage in introducing disconnected maximal strong components in MSC(se1) among those built up from vertices of sv2 .

Secondly, the directed path from the "oldest" to the "youngest" component in the chain --as "cycle in statu nascendi"-- is easily maintained, as is shown by the following analysis.

Suppose that se2 contains an edge that is outgoing from one of the vertices of the youngest maximal strong component in the chain. Such an edge "e" is then transferred from se2 to se1 , and the state of affairs is easily maintained:

1) if e leads to a vertex from sv1 , it can be ignored on account of Theorem 2.

2) if e leads to a vertex from sv2 , the youngest element of the chain can be combined with zero or more next older elements to form the new youngest element of the chain; more precisely: if e leads to a vertex in the youngest element, it can be ignored, if it leads to an older element in the chain, a cycle between strong components has been detected and then Theorem 1a tells us, that a number of the younger elements of the chain have to be combined into a single one, thus reducing the length of the chain, measured in number of elements.

3) if e leads to a vertex from sv3 , that latter vertex is transferred

to sv2 and as new youngest element (a maximal strong component in MSC(se1) all by itself) it is appended to the chain, whose length is increased by one.

If there exists no such edge "e", there are two possibilities. Either the chain is non-empty, but then Theorem 2a tells us, that this maximal strong component of MSC(se1) is a maximal strong component of MSC(se) as well: the youngest element is removed from the chain and its vertices are transferred from sv2 to sv1. Or the chain is empty: if sv3 is not empty, an arbitrary element of sv3 can be transferred to sv2 , otherwise the computation is finished.

In the above degree of detail we can describe our algorithm as follows:

```
se1, se2, sv1, sv2, sv3 := empty, se, empty, empty, sv;
do sv3 ≠ empty → {the chain is empty} .
   transfer a vertex v from sv3 to sv2 and initialize the
        chain with {v};
   do sv2 ≠ empty → {the chain is non-empty}
      do se2 contains an edge starting in a vertex of the youngest
           element of the chain →
              transfer such an edge e from se2 to se1 ;
              if e leads to a vertex v in sv1 → skip
               [] e leads to a vertex v in sv2 → compaction
               [] e leads to a vertex v in sv3 → extend chain and trans-
                                                 fer v from sv3 to sv2
              fi
      od; {the chain is non-empty}
      remove youngest element and transfer its vertices from sv2 to sv1
   od {the chain is again empty}
od
```

Note 1. As soon as vertices are transferred from sv2 to sv1 , their incoming edges (if any) that are still in se2 could be transferred simultaneously from se2 to se1 , but the price for this "advanced" processing --the gain of which is doubtful-- is that we have to be able to select for a given vertex the set of its incoming edges: as the algorithm is described, we only need to find for each vertex its outgoing edges. Hence the above arrangement. (End of note 1.)

Note 2. Termination of the innermost repetition is guaranteed by decrease of the number of edges in se2; termination of the next embracing repetition is guaranteed by decrease of the number of vertices in sv2 ≠ sv3 ; termination of the outer repetition is guaranteed by decrease of the number of vertices in sv3 . The mixed reasoning, sometimes in terms of edges and sometimes in terms of vertices, is a symptom of the non-triviality of the algorithm we are developing. (End of note 2.)

To the degree of detail in which we have described our algorithm, each edge is transferred once from se2 to se1 and each vertex is transferred once from sv3 via sv2 to sv1 : as such our algorithm implies an amount of work linear in the number of edges and vertices. In our next refinement we should try not to spoil that pleasant property, as we would do if, for instance, the test, whether v is in sv1 , sv2 or sv3 --which occurs within the innermost repetition!-- implied a search with a computation time proportional to the number of vertices. The restricted way in which our vertex sets are manipulated, in particular the fact that the vertices enter and leave the chain in last-in-first-out fashion, can be exploited for this purpose.

We consider our vertices consecutively numbered and tabulate the function "rank(v)", where v ranges over all vertex numbers; we assume NV to denote the number of vertices:

rank(v) = 0 means: vertex nr. v is in sv3

rank(v) > 0 means: vertex nr. v is in sv1 ∓ sv2 .

(The sets sv2 and sv1 are, to start with, combined: one of the possible forms of compaction is a skip!)

If nvc equals the "number of vertices in the chain" --i.e. the number of vertices in sv2 -- then

1 ≤ rank(v) ≤ nvc means: vertex v is in sv2

rank(v) ≥ NV + 1 means: vertex v is in sv1 .

All vertices in sv2 will have different rank-values, and as far as rank and nvc are concerned, transferring vertex v from sv3 to sv2 will be coded by "nvc:= nvc + 1; rank:(v)= nvc"

i.e. the vertices in the chain are "ranked" in the order of increasing "age in the chain". The latter convention allows us to represent, how the vertices of sv2 are partitioned in strong components quite efficiently: vertices belonging to the same element of the chain have consecutive values of rank , and for the elements themselves, the rank of their oldest vertex is an increasing function of the element age. Using cc(i) to denote the rank of the oldest vertex of the i-th oldest element of the chain --we have then: cc.dom = the number of elements in the chain-- , as far as rank , nvc and cc are concerned, we can code the alternative construct (combining the first two alternatives) as follows:

```
if rank(v) > 0 → do cc.high > rank(v) → cc:hirem od
 [] rank(v) = 0 → nvc:= nvc + 1; rank:(v)= nvc; cc:hiext(nvc)
fi
```

In the mean time we have somewhat lost trace of the identity of the vertices in the chain: if, for instance, we would like to transfer the vertices of the youngest element of the chain from sv2 to sv1, our current tabulations would force us to scan the function rank for all values of v, such as to find those satisfying $cc.high \leq rank(v) \leq nvc$. We would not like to do that, but thanks to the fact that at least for the vertices in sv2, all values of rank(v) are different, we can also store the inverse function:

for $1 \leq r \leq nvc$: $\quad rank(v) = r \;\Leftrightarrow\; knar(r) = v$

So much for keeping track of the vertices; let us now turn our attention to the edges. The most crucial question with regard to the edges is, of course, the guard of the innermost repetitive construct: "se2 contains an edge starting in a vertex of the youngest element of the chain". That question is answered easily with the aid of a list of edges from se2, outgoing from the vertices of the youngest element of the chain. One of the ways in which the youngest in the chain may change, however, is compaction: in order to maintain that list we, therefore, also need the corresponding lists for the older elements of the chain. Because for those edges we are only interested in the identity of their "target vertex", we introduce as the next part of our chain administration two further array variables --with domain = 0 when the chain is empty-- called "tv" (for "target vertices) and "tvb" (for "tv-bounds").

The domain of tvb will have one point for each element of the chain: its value equals the number of outgoing edges of se2 from vertices of older elements in the chain (the domain of tvb is all the times equal to that one of cc, which also stores one value for each chain element). Each time a new vertex v is transferred from sv3 to sv2, the array tvb is extended at the high end with the value of tv.dom, whereafter tv is extended at the high end with the target vertices of the outgoing edges of v. Denoting that latter operation with "extend tv with targets of v" the whole inner repetition now becomes --taking knar, tv and tvb into account as well--

```
"inner loop":
do tv.dom > tvb.high →
      v, tv:hipop;
      if rank(v) ≥ 0 →
            do cc.high > rank(v) → cc:hirem; tvb:hirem od
       [] rank(v) = 0 →
            nvc:= nvc + 1; rank:(v)= nvc; knar:hiext(v);
            cc:hiext(nvc); tvb:hiext(tv.dom);
            "extend tv with targets of v"
      fi
od
```

We had introduced for vertices v in sv1 the convention: $rank(v) \overset{N}{\geq} NV$. We can make a stronger convention by numbering the maximal strong component from 1 onwards (in the order in which they are detected) and introducing the convention that for a vertex v in sv1 we will have

rank(v) = NV + v's maximal strong component number .

With the variable "strno" (initially = 0), we can now code the

"middle loop":

```
do cc.dom > 0 →
      "inner loop";
      strno:= strno + 1;
      do nvc ≥ cc.high →
            nvc:= nvc - 1; rank:(knar.high)= NV + strno;
            knar:hirem; sv1count:= sv1count + 1
      od;
      cc:hirem; tvb:hirem
od
```

(The variable sv1count --initially = 0-- counts the number of vertices in sv1 : then sv1count = NV will be the criterion for completion of the task.

We assume the vertices numbered from 1 through NV , and the edges to be given by means of two array constants "edge" and "edgeb", such that for $1 \leq i \leq NV$ the values of edge(j) for $edgeb(i) \leq j < edgeb(i + 1)$ give the numbers of the vertices to which the edges outgoing from vertex nr. i lead. We can then code

"extend tv with targets of v ":

```
begin glocon edge, edgeb, v; glovar tv; privar j;
   j vir int := edgeb(v + 1);
   do j > edgeb(v) → j:= j - 1; tv:hiext(edge(j)) od
end
```

The last problem to be solved is the selection of an arbitrary vertex v from sv3 for the initialization of the chain. If each time the search would start at vertex nr. 1 , computation time could be proportional to NV^2 , but again this can be avoided by taking a relation outside the repetition and introducing at the outer level a variable "cand" (initially = 1) with the property:

sv3 contains no vertex v with $v < cand$.

```
begin glocon edge, edgeb, NV; virvar rank; privar sv1count, cand, strno;
   rank vir int array := (1); do rank.dom ≠ NV → rank:hiext(0) od;
   sv1count vir int, cand vir int, strno vir int := 0, 1, 0;
   do sv1count ≠ NV →
      begin glocon edge, edgeb, NV; glovar rank, sv1count, cand, strno;
         privar v, cc, tv, tvb, knar, nvc;
         do rank(cand) ≠ 0 → cand:= cand + 1 od; v vir int := cand;
         nvc vir int := 1; rank:(v)= 1; knar vir int array := (1, v);
         cc vir int array := (1, 1); tvb vir int array := (1, 0);
         tv vir int array := (1);
         "extend  tv  with targets of  v ";
         "middle loop"
      end
   od
end
```

Note 1. A very similar algorithm has been developed independently by Robert Tarjan. (End of note 1.)

Note 2. In retrospect we see that the variable "nvc" is superfluous, because nvc = knar.dom . (End of note 2.)

Note 3. The operation "extend tv with the targets of v " is used twice. (End of note 3.)

Remark 1. The reader will have noticed that in this example the actual code development took place in a different order than in the development of the program for the convex hull in three dimensions. The reason is --I think-- the following. In the case of the convex hull, the representation had already been investigated very carefully as part of the logical analysis of the problem. In this example the logical analysis had been largely completed when we faced the task of selecting a representation that would admit an efficient execution of the algorithm we had in mind. It is then natural to focus one's attention on the most crucial part first, i.e. the innermost loop. (End of remark 1.)

Remark 2. It is worth noticing the various steps in which we arrived at our solution. In the first stage our main concern has been to process each edge only once, forgetting for the time being about the dependance of the computation time on the number of vertices. This is fully correct, because, in general, the number of edges can be expected to be an order of magnitude larger than the number of vertices. (As a matter of fact, my first solution for this problem --not recorded in this chapter-- was linear in the number of edges but quadratic in the number of vertices.) It was only in the second stage that we started to worry about linear dependance on the number of vertices as well. How effective this "separation of concerns" has been is strikingly illustrated by the fact that in that second stage, graph theory did no longer enter our considerations at all! (End of remark 2.)