Copyright Notice

The following manuscript

EWD 376: Finding the maximum strong components in a directed graph

is held in copyright by Springer-Verlag New York.

The manuscript was published as pages 22–30 of

Edsger W. Dijkstra, *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag, 1982. ISBN 0–387–90652–5.

Reproduced with permission from Springer-Verlag New York.
Any further reproduction is strictly prohibited.

Finding the maximum strong components in a directed graph.

This essay records an exercise in orderly program composition. The record is not completely truthful in the sense that prior to its writing some thinking without pencil and paper was done. As a result, the following text contains a few "surprises" in the sense that suggestions are made without an elaborate heuristic justification. When I noticed myself doing so, some heuristic justification has been added afterwards. The moral of all this is: in case of surprise, please go on reading!

Given a set of nodes and a set of directed arcs leading from a node to a node, it is requested to partition the set of nodes into maximal strong components. A strong component is a set of nodes such that the arcs between them provide a path from any node of the set to any node of the set; a single node 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 nodes can be added.

We shall use the acronym "sa" for a set of arcs, the acronym "sn" for a set of nodes. Our final answer is a partitioning, that is a set of sets of nodes with empty intersections: for that latter object we shall use the acronym "ssn". Similarly, when the need arises, we shall use the acronym "ssa" for a set of sets of arcs with empty intersection. (Note added while typing out the manuscript: this need has not arisen.)

Let "sn" be the given set of nodes, let "sa" be the given set of arcs. Let the final value of "ssn" be the desired answer. We then write the desired final relation as

ssn = MSC(sa) (1)

where MSC, the set of Maximal Strong Components, is regarded for constant sn as a function of the set of arcs sa.

We want to inspect the arcs one by one (in a suitable order still to be chosen), i.e. we introduce two disjoint subsets of sa, viz. sa1 and sa2, such that

sa = sa1 + sa2 (2)

where sa1 comprises the arcs inspected (initially empty, finally = sa) and sa2 the arcs uninspected (initially = sa, finally empty).

Similarly, we want to build up the final value of ssn. We shall do so by maintaining the invariant relation

ssn + ssn1 = MSC(sa1) (3)

Here each node of sn will either occur in an element of ssn or in an element of ssn1, but never in both. (Besides that we can, as will be shown later, restrict ourselves to ssn1-values being sets of sets of single nodes.) The following idea was underlying the introduction of ssn1: ssn is a set of maximal strong components, for which -we write an algorithm for a sequential machine!- we may expect to establish one after the other that they will occur as element of the final value of ssn. Our aim is that at any moment in time, ssn will only contain elements of its final value: they are the maximal strong component definitely found. Then we need ssn1 for the remaining nodes.

The initialization corresponding to sa1 = empty is ssn = empty and ssn1 with each node of sn in a separate element of ssn1. When we succeed in establishing

ssn1 = empty <u>and</u> sa2 = empty (4)

under invariance of (3), the desired relation (1) has been established, as the second term of (4) implies on account of (2) that sa = sa1.

We have not established yet the relation between the way in which the nodes are divided over ssn and ssn1 on the one hand and the arcs over sa1 and sa2 on the other. We shall maintain the following relations (5) and (6):

each arc originating in a node of ssn will be in sa1 (5)

each arc terminating in a node of ssn1 will be in sa2 . (6)

Relations (5) and (6) are compatible with the initial situation: because ssn = empty, there will be no arcs originating in a node of ssn and therefore sa1 can be empty (i.e. (5) is not violated) and because ssn1 comprises all nodes, all arcs should be in sa2, in accordance with the initial condition sa2 = sa (i.e. (6) is satisfied).

Relations (5) and (6) are also compatible with the final situation: because then ssn will comprise all nodes, all arcs must be in sa1, in accordance with sa1 = sa (i.e. (5) is satisfied) while (6) is satisfied because then both ssn1 and sa2 will be empty (see (4)).

We observe that, because sa1 and sa2 have an empty intersection, there will be no arcs originating in a node of ssn and terminating in a node of ssn1. On the other hand, an arc originating in a node of ssn1 and terminating in a node of ssn may be either in sa1 or in sa2.

The structure of our program becomes, if we want to apply the fundamental invariance theorem for loops:

```
sa1:= empty; sa2:= sa;
ssn:= empty; ssn1:= "the set of all single node sets";
while ssn1 ≠ empty or sa2 ≠ empty do
      "transfer arc(s) from sa2 to sa1" and/or
      "transfer node(s) from ssn1 to ssn"
       under invariance of (3), (5) and (6)
                                          od
```

Relation (5) allows us to simplify the last boolean expression: ssn1 = empty implies that all nodes are in ssn; this implies that all arcs are in sa1, which implies that sa2 = empty. Therefore it can be simplified to

```
while ssn1 ≠ empty do .
```

Relations (5) and (6), which may have come as a surprise, have been suggested by

<u>Theorem 1</u>. When the set of nodes are subdivided into two sets nsA and nsB, such that there are no arcs originating in a node of nsA and terminating in a node of nsB, then the set of strong components is unchanged when the arcs (if any) originating in a node of nsB and terminating in a node of nsA are removed and, secondly, no strong component comprises nodes from both sets.

Here the nodes in ssn play the role of those in snA and Theorem 1 tells us that the maximal strong components they will give rise to cannot depend on the arcs still in sa2. Therefore they can only depend on the arcs in sa1 that have already been inspected. As a result each element (i.e. a maximal strong component) of an intermediate value of ssn will be an element of its final value.

In order to detail the repeatable statement we introduce a chain of strong components (a chain of sets of nodes), called "csn"; empty at the beginning and at the end of the repeatable statement. The transfer of a node from ssn1 to ssn will take place in two steps: first the node will be transferred (individually) from ssn1 to csn, at a later stage the node will be transferred (together with all the nodes of the same maximal strong component) from csn to ssn.

The strong components in csn are so by virtue of arcs of sa1 and their chaining is performed by arcs of sa1, more precisely

two successive strong components in csn are connected by <u>one</u> arc from sa1 originating in a node of the predecessor and terminating in a node of the successor (7)

no arc in sa1 will originate at a node of an element of csn and terminate at a node of a preceding element in csn . (8)

The chain csn has been introduced as a tool for the searching for cycles, an activity that is suggested by

<u>Theorem 2</u>. When a number of strong components can be connected via a cyclic path, they belong to the same maximal strong component.

This theorem suggests that we try to extend the chain at one end: whenever we encounter an arc leading from its end element to a preceding element in the chain, from and including that preceding element up to and including the terminal element can be combined to form the new terminal element. We shall call this operation "combine end elements of csn"; its purpose is to restore the validity of (8).

When the chain csn is non-empty, we investigate whether sa2 contains an arc f having its origin in (one of the nodes of) the terminal element of csn.

If such an arc f points to one of the nodes in ssn, it can be ignored (on account of Theorem 1).

If such an arc f points to a node in the terminal element of csn, it can be ignored as well -we knew already that the nodes in this terminal element formed a strong component.

If such an arc points to (a node in) a preceding element of csn, the end elements of csn are combined.

If such an arc leads to a node in ssn1, that node is appended to the chain and will form, all by itself, the new terminal element of csn.

In all four cases the arc f is transferred from sa2 to sa1.

If no such arc exists, the terminal element of the chain must be a maximal strong component of the final graph, will be removed from csn and added to ssn, which now grows by one element. This conclusion, again, is justified by Theorem 1. (Note. Here Theorem 1 is applied twice: the terminal node is a maximal strong component because it has no outgoing arcs in the reduced graph that we get by removing all arcs leading back to a node of ssn after it has been established that ssn already contains maximal strong components for the total graph.)

The structure of the repeatable statement -only starting when the chain csn = empty and ssn1 ≠ empty- can be the following:

```
transfer an arbitrary element of ssn1 and append it to an
initially empty chain csn;
while csn ≠ empty do
   if sa2 contains no arc f originating in a node of csn's terminal
      element
    then transfer csn's terminal element to ssn
    else transfer such  an  arc  f from sa2 to sa1;
         if f terminates in (a node of) an element of ssn1
           then transfer that element from ssn1 by appending it to csn
           else if f leads to (a node of) a preceding element of csn
                  then combine end elements of csn
                fi
         fi
   fi
od
```

We have now to choose a way for representing the information. It is assumed that the nodes are numbered from 1 through N. Because we intend to chain nodes, it is a wise precaution to add "a virtual node" with number 0.

In the representation of our sets of nodes we can exploit the fact that we know that the elements of ssn1 are single node sets. In ssn and csn our elements are strong components, in csn we can number them from +1 upwards, in ssn we can number them from -1 downwards and thus we come to the following representation with an integer array sn[0:N]

sn[i] > 0 means: node i is a member of element sn[i] of csn
sn[i] < 0 means: node i is a member of element sn[i] of ssn
sn[i] = 0 means: node i is (a node of) an element of ssn1
sn[0] = 0 .

In order to scan nodes we introduce for nodes in csn or ssn an integer array pc[1:N], where for node i in one of the two sets of sets

pc[i] = j means: with respect to node i, node j is the next oldest node in the same set of sets; when j = 0, node i is its oldest node.

In order to be able to trace these pc-chains we introduce two handles:

yc = the number of the youngest node in csn; when csn = empty, yc = 0
ys = the number of the youngest node in ssn; when ssn = empty, ys = 0.

In order to speed up the search for an arbitrary node in ssn1 for the initialization of csn, we introduce the integer k, such that ssn1 contains no nodes with a number < k.

Further we introduce, in order to be able to fix the ordinal number of a new element
ec = the number of elements in csn
es = the number of elements in ssn
and, in order to decide whether ssn1 is empty
es1 = the number of elements in ssn1.

In our program we have to establish whether sa2 contains an arc f originating from the terminal element of csn. We do so by investigating the nodes of the terminal element and on account of the pc-chaining we do so in order of increasing age in csn. Because quite a number of nodes may be a member of the terminal element it seems a bit wasteful in time to start this search always at the youngest node and therefore we introduce

yun = the number of "the youngest possibly unexhausted node" i.e. sa2 contains no arcs originating in a node of csn younger than nr. yun (if any). Again, in the extreme case, yun may get the value 0.

Our algorithm presupposes that for each node we can find "its outgoing arcs". We therefore assume that the arcs are sorted in the order of increasing starting node and that in that order their terminal nodes are listed in the global integer array $t[1:\text{number of arcs}]$ while the boundaries are given by the integer array $b[0:N]$, such that $b[0]= 0$, $b[N] = \text{number of arcs}$ and the nodes at which the arcs originating at node i terminate will be $t[k]$, with k ranging

$$b[i-1] < k \leq b[i] \quad .$$

For the representation of the partitioning sa = sa1 + sa2 we introduce

<u>integer</u> <u>array</u> $c[0:N]$

such that all arcs originating in node i and belonging to sa1 will have an ordinal number k satisfying

$$b[i-1] < k \leq c[i]$$

and those in sa2 a k satisfying

$$c[i] < k \leq b[i] \quad .$$

We assume $c[0] = 0$ for the sake of safety (i.e. sa2 contains no arcs originating from the virtual node).

In the following program the variable ft is used to identify the terminal node of arc f, while the variable h is used for a wild collection of short range purposes. I know that this is a poor style: I too have my weak moments!

```
begin integer array sn, c [0 : N], pc [1 : N];
      integer yc, ys, ec, es, es1, yun, h, ft, k;
   {initialize sa1 and sa2}
      c[0]:= 0; h:= 0; while h < N do h:= h + 1; c[h]:= b[h-1] od;
   {initialize ssn and ssn1}
      h:= 0; while h ≤ N do sn[h]:= 0; h:= h + 1 od;
      ys:= 0; es:= 0; es1:= N; k:= 1;
      while es1 > 0
      do
      {search for a node k in ssn1}
        while sn[k] ≠ 0 do k:= k + 1 od;
      {remove it from ssn1 and initialize csn with node k}
        es1:= es1 - 1; sn[k]:= 1;
        pc[k]:= 0; yc:= k; ec:= 1; yun:= k;
      {note that at this moment node k is oldest and youngest and youngest
       possibly unexhausted node of csn}
        while ec > 0
        do
        {search for the youngest unexhausted node of the terminal element
         of csn}
          while sn[yun] = ec and c[yun] ≥ b[yun] do  yun:= pc[yun] od
        {this loop will certainly terminate, possibly with yun = 0};
          if sn[yun] ≠ ec
            then {there is no arc f in sa2 originating in the terminal
                 element nr. ec of csn and therefore this terminal element
                 will be transported to ssn}
                   es:= es + 1;
                   while sn[yc] = ec
                   do sn[yc]:= - es; h:= pc[yc]; pc[yc]:= ys;
                      ys:= yc; yc:= h
                   od;
                   ec:= ec - 1; yun:= yc
            else { c[yun] < b[yun], therefore the next arc originating at
                 node nr. yun will be transferred from sa2 to sa1; this is arc f}
                   c[yun]:= c[yun] + 1; ft:= t[c[yun]]; h:= sn[ft];
              {now ft is the terminal node of arc f and h = sn[ft] to save
               dynamically a few subscriptions!}
```

```
            if h = 0
              then {node ft has to be removed from ssn1 and to be
                  attached to csn}
                    es1:= es1 - 1; ec:= ec + 1; sn[ft]:= ec;
                    pc[ft]:= yc; yc:= ft; yun:= yc
              else if 0 < h and h < ec
                     then {ft is a node of the non-terminal element
                         nr. h of csn, with which the younger elements
                         have to be combined}
                           ec:= h
                         {this ends the use of h as h = sn[ft]};
                           h:= yc; while sn[h] > ec
                                     do sn[h]:= ec; h:= pc[h]
                                     od
                         { note that in combining, pc, yc and yun can
                          remain unchanged}
                     else {arc f points either to csn's terminal
                          element or to an element of ssn; in either
                          case it can be ignored}
                     fi
            fi {the case that arc f existed has been dealt with}
        fi {csn's terminal element has been inspected}
      od {csn is again empty}
   od {ssn1 is empty, the computation has been done};
 {print the results; the maximal strong components appear numbered in
  decreasing order}
   while es > 0
   do newline; printtext("maximal strong component nr.");
      printvalue(es); printtext("consists of the nodes:");
      while sn[ys] = - es do printvalue(ys); ys:= pc[ys] od;
      es:= es - 1
   od
end
```

Concluding remarks.

In order to avoid the usual misunderstandings it might be a good thing to point out, once again, that the approach that has been illustrated in this exercise does not pretend to be an infallible cure against fallibility. We have tried two things: we have tried to develop a program in a way that leads to a higher confidence level than the one that can be reached when the designer "rushes into coding" and we have tried to make the reader share our conviction -strengthened by the above experience!- that the simultaneous development of the correctness proof gives indeed a strong heuristic guidance in the process of shaping the program.

As the reader will have noticed we have not spent a single word of explanation on the repeatable statement of the small innermost loops. I think that this is in accordance with normal mathematical practice: the reasoning has to be broken down in steps so small that they can be made "in confidence" and that a more detailed proof, a more detailed justification could be given when they are challenged, but that that should not be done without compelling reason. We should not waste our time on trivia!

The situation at the innermost loops, where we deal with quite standard coding techniques, is quite different from the situation at the outermost levels where we have to manipulate with concepts and relations cooked up and discovered for the specific purpose of solving this specific problem: it is at the latter level that the greater explicitness seems most urgently needed. Also, it is in that part of the analysis and synthesis that the most heavy demands are made upon the programmer's ability to express himself effectively.

Finally we draw attention to the fact that we did not need a single example to explain what we were talking about or (even worse!) to discover what the program should do. And this, of course, is as it should have been.

Acknowledgements.

We express our gratitude towards J.A.G.M.Kerbosch and J.C.Wortmann for bringing this problem to our attention and thereby presenting the challenge.

30th May 1973

Edsger W.Dijkstra