The problem of the most isolated villages.

We consider n villages $(n > 1)$, numbered from 0 through $n - 1$; for $0 \le i < n$ and $0 \le j < n$, a computable function $f(i, j)$ is given, satisfying for some given positive constant M:

$$\text{for } i \ne j : \quad 0 < f(i, j) < M$$
$$\text{for } i = j : \quad f(i, j) = M \quad .$$

For the i-th village, its isolation degree "id(i)" is given by

$$id(i) = \underset{j \ne i}{\text{minimum}}\ f(i, j) = \underset{j}{\text{minimum}}\ f(i, j)$$

(Here $f(i, j)$ can be interpreted as the distance from i to j; the rule $f(i, i) = M$ has been added for the purpose of the above simplification.)

We are requested to determine the set of maximally isolated villages, i.e. the set of all values of k such that

$$(\underline{A}\ h\colon 0 \le h < n\colon id(h) \le id(k)) \quad .$$

The program is expected to deliver this set of values as

✱ $$miv(miv.lob), \ldots , miv(miv.hib)$$

Note that eventually all values $1 \le miv.dom \le n$ are possible.

A very simple and straightforward program computes the n isolation degrees in succession and keeps track of their maximum value found thus far. On account of the bounds for $f(i, j)$ we can take as the minimum of an empty set the value M and as the maximum of an empty set 0.

```
begin glocon n, M; virvar miv; privar max, i;
   miv vir int array := (0); max vir int, i vir int := 0, 0;
   do i ≠ n →
   begin glocon n, M; glovar miv, max, i; privar min, j;
      min vir int, j vir int := M, 0;
      do j ≠ n →
            do f(i, j) < min → min:= f(i, j) od;
            j:= j + 1
      od {min = id(i)};
      if max > min → skip
       ▯ max = min → miv:hiext(i)
       ▯ max < min → miv:= (0, i); max:= min
      fi;
      i:= i + 1
   end
   od
end
```

The above is a very unsophisticated program: in the innermost loop the value of min is monotonically non-increasing in time, and the following alternative construct will react equivalently to <u>any</u> value of min satisfying $\text{min} < \text{max}$. Combining these two observations we conclude that there is only a point in continuing the innermost repetition as long as $\text{min} \geq \text{max}$. We can replace the line "<u>do</u> j ≠ n →" therefore by

"<u>do</u> j ≠ n <u>and</u> min ≥ max →"

and the assertion after the corresponding <u>od</u> by

$$\{\text{id}(i) \leq \text{min} < \text{max} \ \underline{\text{or}}\ \text{id}(i) = \text{min} \geq \text{max}\} \quad .$$

Let us call the above modification "Optimization 1".

A very different optimization is possible if it is given that

$$f(i, j) = f(j, i)$$

and --because the computation of f is assumed to be time-consuming-- it is requested never to compute $f(i, j)$ for such values of the argument that $f(j, i)$ has already been computed. Starting from our original program we can achieve that for each unordered argument pair the corresponding f-value will only be computed once by initializing j each time with $i + 1$ instead of with 0 --only scannig the upper triangle of the symmetric distance matrix, so to speak-- . The program is then only guaranteed to compute min correctly provided that we initialize min instead of with M with

$$\underset{0 \le h < i}{\text{minimum}} \; f(i, h) \quad .$$

This can be catered for by introducing an array , b say, such that for k satisfying $i \le k < n$:

$$\text{for } i = 0 \;\; : \;\; b(k) = M$$

$$\text{for } i > 0 \;\; :: \;\; b(k) = \underset{0 \le h < i}{\text{minimum}} \; f(k, h)$$

(In words: $b(k)$ is the minimum distance connecting village k that has been computed thus far.)

The result of Optimization 2 is also fairly straightforward.

```
begin glocon n, M; virvar miv; privar max, i, b;
   miv vir int array := (0); max vir int, i vir int := 0, 0;
   b vir int array := (0); do b.dom ≠ n → b:hiext(M) od;
   end {all virgin variables at outer level are initialized};
   do i ≠ n →
   begin glocon n; glovar miv, max, i, b; privar min, j;
      min vir int, b:lopop; j vir int := i + 1;
      do j ≠ n →
      begin glocon i; glovar min, j, b; privar ff;
         ff vir int := f(i, j);
         do ff < min → min:= ff od;
         do ff < b(j) → b:(j)= ff od
         j:= j + 1
      end
      od {min = id(i)};
      if max > min → skip
       ▯ max = min → miv:hiext(i)
       ▯ max < min → miv:= (0, i); max:= min
      fi;
      i:= i + 1
   end
   od
end
```

To try to combine these two optimizations presents a problem: in Optimization 1 the scanning of a row of the distance matrix is aborted if min has become small enough, in Optimization 2, however, the scanning of the row is also the scanning of a column and that is done to keep the values of $b(k)$ up to date. Let us apply Optimization 1 and replace the line "<u>do</u> $j \neq n \rightarrow$" by

"<u>do</u> $j \neq n$ <u>and</u> $min \geq max \rightarrow$" .

The innermost loop can now terminate with $j < n$; the values $b(k)$ with $j \leq k < n$, for which updating is still of possible interest are now the ones with $b(k) \geq max$, the other ones are already small enough. The following insertion will do the job:

```
do j ≠ n →
    if b(j) < max → j:= j + 1
     ▯ b(j) ≥ max →
     begin glocon i; glovar j, b; privar ff;
        ff vir int := f(i, j);
        do ff < b(j) → b:(j)= ff od;
        j:= j + 1
     end
    fi
od                .
```

The best place for this insertion is immediately preceding "i:= i + 1", but after the adjustment of max : the higher max, the larger the probability that a $b(k)$ does not need anymore adjustments.

* * *

The two optimizations that we have combined are of a vastly different nature: Optimization 2 is just "avoiding redoing work known to have been done", and its effectiveness is known a priori. Optimization 1, however, is a strategy, whose effectiveness depends on the unknown values of f: it it just one of the many possible strategies in the same vein.

We are looking for those rows of the distance matric whose minimum element value S exceeds the minimum elements of the remaining rows and the idea of Optimization 1 is that for that purpose we do not need to compute for the remaining rows the actual minimum if we can find for each row an upper bound B_i for its minimum, such that $B_i < S$. In an intermediate stage of the computation, for some row(s) the minimum S is known because all its/their elements have been computed; for other rows we only know an upper bouød B_i . And now the strategic freedom is quite clear: do we first compute the smallest number of additional matrix elements still needed to determine a new minimum, in the hope that it will be larger than the minimum we had and, therefore, may exceed a few more B's? Or do we first compute unknown elements in rows with a high B in the hope of cheaply decreasing that upper bound? Or any mixture?

My original version combining the two strategies postponed the "updating of the remaining b(k)" somewhat longer, in the hope that in the mean time max would have grown still further, but whether it was a more efficient program than the one published in this chapter is subject to doubt: it was certainly more complicated, needing yet another array for storing a sequence of village numbers. The published version was only discovered when writing this chapter.

In re~~p~~trospect I consider my ingenuity spent on my original program as wasted: if it was "more efficient" it could only be so "on the average". But on what average? Such an average is only defined provided that we postulate --quite arbitrarily!-- a probability distribution for the distance matrix $f(i, j)$. On the other hand it was not my intention to tailor the algorithm to s specific subclass of distance matrices!

The moral of the story is that, in making a general program, we should hesitate to yield to the temptation to incorporate the smart strategy that would improve the performance in cases that might never occur, if such incorporation complicates the program notably: simplicity of the program is a less ambiguous target. (The problem is that we are often so proud of our smart strategies that it hurts to abandon them.)

Remark. Our final program combines two ideas and we have found it by first considering --as "stepping stones", so to speak-- two programs, each incorporating one of them, but not the other. In many instances I found such stepping stones most helpful. (End of remark.)