INSTITUT FÜR PLASMAPHYSIK

GARCHING BEI MÜNCHEN

Subroutines storage overlapping

by

K. Hain and F. Hertweck

IPP-6/6 August 1963

A method for minimizing the storage used by a program is presented.

Die nachstehende Arbeit wurde im Rahmen des Vertrages zwischen dem Institut für Plasmaphysik GmbH und der Europäischen Atomgemeinschaft über die Zusammenarbeit auf dem Gebiete der Plasmaphysik durchgeführt.

In this paper we present a method to minimize the storage locations needed by a program consisting of a "main" program and a number of "subroutines". The method applies only to the storage locations for "own" variables, i.e. for variables defined only for a certain subroutine. Besides these "own" variables there are the parameters defined in the calling program. The point is that the contents of the storage locations for "own" variables must not be destroyed by any subroutine called by the considered one. On the other hand, independent subroutines may share their storage, so by this overlapping of storage the number of storage locations needed can be reduced.

This procedure of storage allocation is a static one; that means the storage allocation is performed when the whole program is brought into storage. It is assumed that the dimensions of all arrays are known. It seems reasonable to introduce such a procedure instead of "dynamic" storage allocation during execution because of the time consumption and the size of the program in the latter case.

One could also consider of a "half-dynamic" storage allocation. This is reasonable in the case that the program has to be executed several times independently with different parameter sets. The dimensions of some arrays may be different in these "runs". Now if the program with all arrays fixed to their respective maximum size would not fit into storage, while the actual arrays for each run would, then the "half-dynamic" storage allocation would be of great help. So a certain "storage allocation part" of the program has to be performed and the whole program has to be relocated before execution.

We define a program to consist of a main program and a certain (finite) number N of subroutines. It is understood that there is always one and only one main program.

Subroutines are called by at least one other subroutine and/or the main program, whereas the main program is not called by any subroutine. Further the program is assumed to be non-cyclic in the following sense: No subroutine can be called by itself (i.e. no recursive calling of subroutines) nor are there cycles in a sequence of subroutines so that a subroutine is indirectly called by itself. This implies that there is at least one subroutine in the program, which does not call any other subroutine. Such a subroutine we shall call an elementary subroutine.

To determine whether there are cycles in a program or not, we can proceed as follows: We start with the main program and select one of the subroutines which are called by the main program say the subroutine S_1. Then we take one of the subroutines which are called by S_1, say S_2 and so forth. If this sequence ends with an elementary subroutine we call this sequence non-cyclic. If the same subroutine occurs at least twice in the same sequence, we call this sequence cyclic. If all possible sequences in a program are non-cyclic, we call the program non-cyclic; if there is only one cyclic sequence, the whole program is cyclic.

We may add the remark that in a non-cyclic program the length of a sequence is always $\leq$ N+1, the total number of subroutines plus the main program. If the program is cyclic, we may continue the sequence ad infinitum with the subroutines in the cycle appearing periodically. Hence it is sufficient to prove that all possible sequences are of length $\leq$ N+1 to assure the non-cyclic property of the program.

To each subroutine may be attributed an integer called the level of the subroutine, according to the following rules:

(1) The main program has level 0

(2) A subroutine of level L, $L \geq 0$, may call only subroutines of level $L' > L$, and, conversely, a subroutine of level L may be called only by subroutines of level $\hat{L} < L$

(3) The level of a subroutine is the smallest integer consistent with rule (2)

We now can state the following

Lemma: In a non-cyclic program there is always one uniquely defined level for each subroutine.

To prove this, we first define the transfer matrix T_{ik} (i denoting the rows and k the columns) by the following rule: Consider the main program and the subroutines labelled by the N+1 integers 0,1,...N; without restriction we can make the main program to have label 0. Now

$$T_{ik} = \begin{cases} 1 & \text{if subroutine i calls subroutine k} \\ 0 & \text{else} \end{cases}$$

The dimensions of T_{ik} are i=0,N, and k=0,N+1. By this definition, the rows and columns have the following meaning: For any i the row gives all subroutines which are called by subroutine i, and for any k the column gives all subroutines which call subroutine k. In column N+1 we shall note the level of the subroutines.

The first column of this matrix contains only zeros, because, by definition, the main program has the property that it is not called by any subroutine.
This implies that $\sum_{i=0}^{N} T_{i0} = 0$.

Now we attribute level 1 to the subroutines which appear in the first row. So, consistent with rule 2, all subroutines called by the main program have level 1. At the same time

condition (3)is fulfilled. We note the levels in column N+1 of the matrix T_{ik} , so

$$\text{level of subroutine } i \equiv Li = T_{i,N+1}$$

Now, for all i = 1...,N for which L_i = L we select the subroutines k called by subroutine i. Their levels must be greater than L_i = L (rule 2). The smallest integer satisfying this condition is L + 1 (rule 3). We note these levels in $T_{k,N+1}$. If we have arrived at i=N we repeat this procedure with L incremented by 1, and so on. Because the program is assumed to be non-cyclic, we will find subroutines i for which all T_{ik} =0, k=1,...,N. These are the elementary subroutines. After a certain finite number of steps we have for all i=1,...,N, and L =$L^* \leq$ N only elementary subroutines. Because if this procedure were not finite the level of at least one subroutine would grow unlimitedly. This is possible only if the program is cyclic. It is also clear that $L^* \leq$ N, the equal sign being valid if and only if there is only one possible sequence in the program with every subroutine calling only one other. Because every subroutine of a program must be called (else it would not belong to the program), it is, by this procedure, assured that every subroutine will get a level. It is unique because all numbers satisfying condition (2) have a unique lower limit.

Corollary:

There exists at least one subroutine in every level. To prove this statement let us assume there are some subroutines in level L but no subroutines in level L+1. This is possible if $L=L^*$ - then the subroutines of level L are all elementary. If they are not, at least one of them calls a subroutine of level $L' >$ L+1. But in that case we can reduce the levels of all these subroutines to L+1 without violating rule (2). Hence, before doing this, rule 3 was violated, and there must be at least one subroutine in level L+1.

We now state the

Lemma: (a) In a non-cyclic program all subroutines of a sequence may not share their working storage areas.
(b) This must hold for all sequences of the program.
(c) Different subroutines in different sequences may share their working storage.

(a) follows immediately from the definition of a sequence. If (b) were not necessary, one could construct a case in contradiction to (a). (a) and (b) are also sufficient as criterion for proper storage allocation, because a subroutine is only called inside a sequence and all sequences are considered by (b). (c) is true because two different subroutines in two different sequences cannot call each other or else they would belong to the same sequence.

We shall now adopt the following procedure for storage allocation. Let us denote by S_i the number of storage locations needed for subroutine i, i=o,1,...,N (i=o is for the main program), by U_i the first used storage locations of, and by F_i the first free storage location behind, subroutine i.

For the main program we have

$$U_o = \text{first location of working storage area}$$

$$F_o = U_o + S_o$$

Then we have for level 1 simply

$$\left.\begin{aligned} U_{i_1} &= F_o \\ F_{i_1} &= U_{i_1} + S_{i_1} = F_o + S_{i_1} \end{aligned}\right\} \quad \text{all } i_1$$

where i_1 are all subroutines of level 1. Let us assume that U_i and F_i for all subroutines of all levels $L' < L$ are known. For every subroutine i_L of level L we find with the help of the transfer matrix all subroutines k^* which call subroutine i_L. Then it is sufficient for the independence of the storage of subroutine i_L of all calling subroutines k^* to make

$$\left.\begin{aligned} U_{i_L} &= \text{MAX} \quad (F_{k_*} \text{ , all } k^* \text{ for which } T_{k^*,\, i} = 1) \\ F_i &= U_i + S_i \end{aligned}\right\} \text{all } i_L$$

So we have determined the storage needs for all subroutines of level L and can proceed to level L+1,L+2,...,L*.

The storage needed by the whole program is given by

$$S_{total} = MAX \quad (F_{iL}, \text{ all } i_L \quad) - U_o$$

By the procedure as explained above the storage needs for a program are as far as possible minimized, by only considering the logical connection of the subroutines of a program at the time the program is brought into storage. If during execution a subroutine is not called at all, a certain amount of storage locations may be wasted. On the other hand, this could be avoided only by a dynamic storage relocation, which has disadvantages of its own. This procedure of storage allocation at loading time is relatively simple and will not consume much computing time.

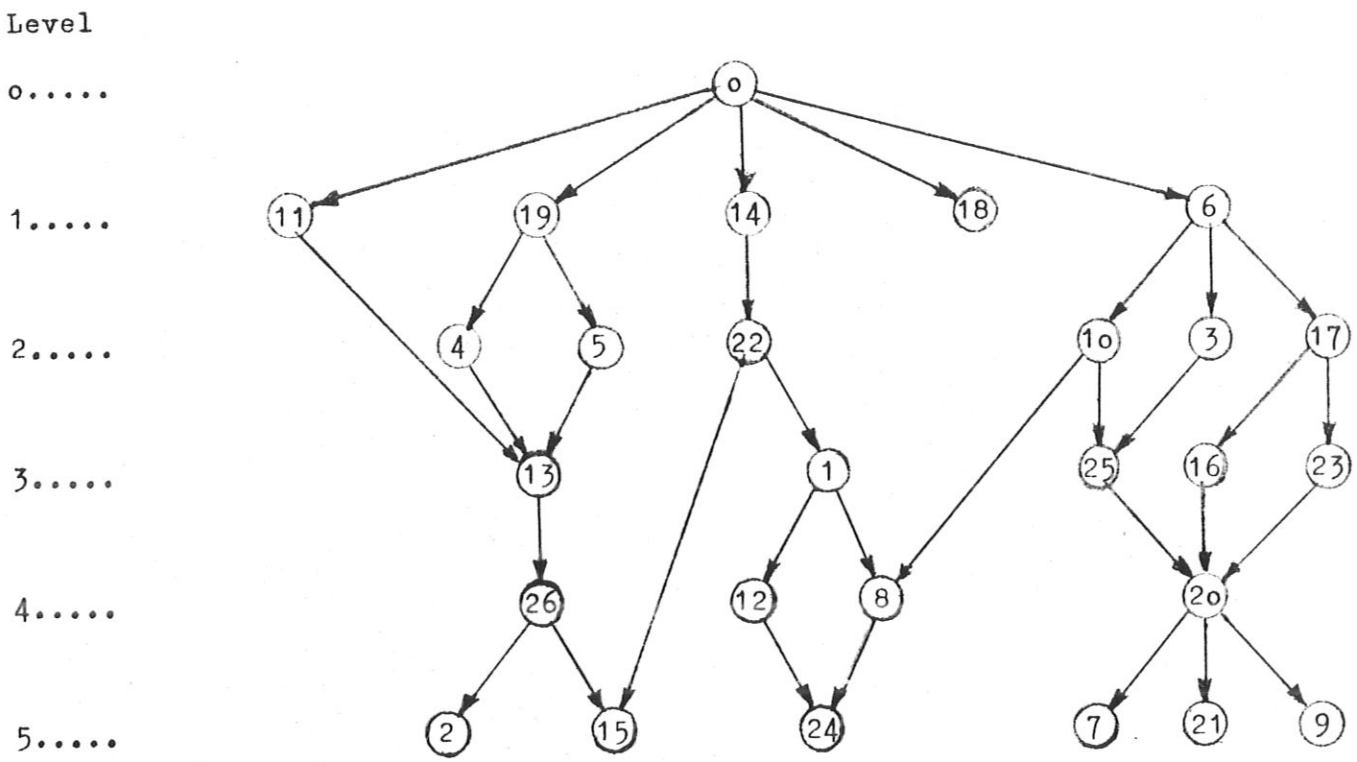
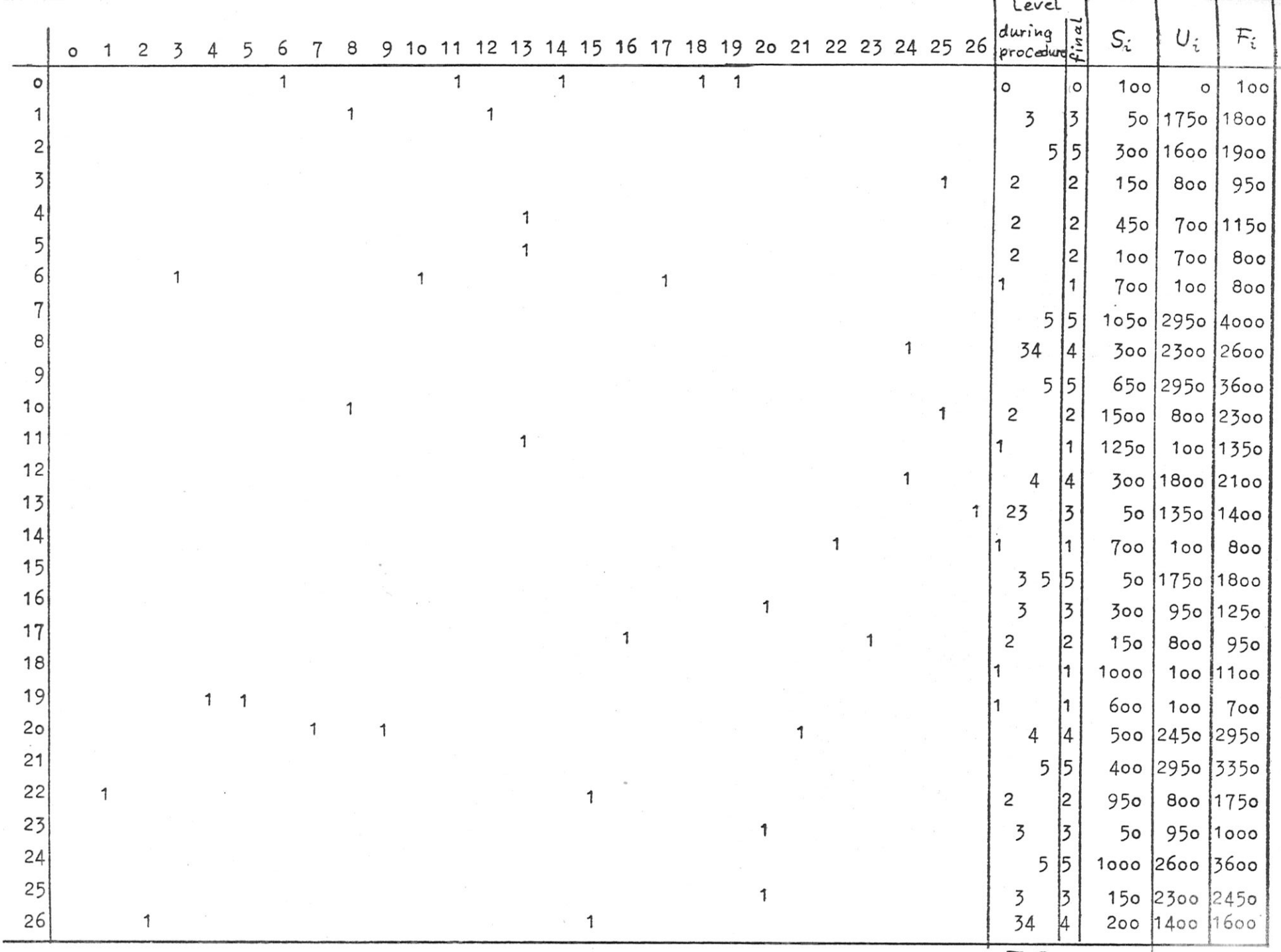


Fig. 1. An example of a program with a main program (label 0) and a certain number of subroutines (labelled arbitrarily from 1 to 26).



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	Level during procedure	final	S_i	U_i	F_i
0							1					1			1				1	1								0	0	100	0	100
1									1				1															3	3	50	1750	1800
2																											5	5	5	300	1600	1900
3																										1		2	2	150	800	950
4														1														2	2	450	700	1150
5														1														2	2	100	700	800
6				1							1							1										1	1	700	100	800
7																										1		5	5	1050	2950	4000
8																			1									34	4	300	2300	2600
9																											5	5	5	650	2950	3600
10									1																	1		2	2	1500	800	2300
11														1														1	1	1250	100	1350
12																									1			4	4	300	1800	2100
13																											1	23	3	50	1350	1400
14																							1					1	1	700	100	800
15																												3 5	5	50	1750	1800
16																					1							3	3	300	950	1250
17																	1							1				2	2	150	800	950
18																												1	1	1000	100	1100
19					1	1																						1	1	600	100	700
20								1		1												1						4	4	500	2450	2950
21																												5	5	400	2950	3350
22		1														1												2	2	950	800	1750
23																					1							3	3	50	950	1000
24																												5	5	1000	2600	3600
25																					1							3	3	150	2300	2450
26			1													1												34	4	200	1400	1600

$\sum S_i = 13000$ Max $F_i = 4000$

Fig. 2. The transfer matrix of the program shown in Fig. 1. The last column is split up into the different states after the

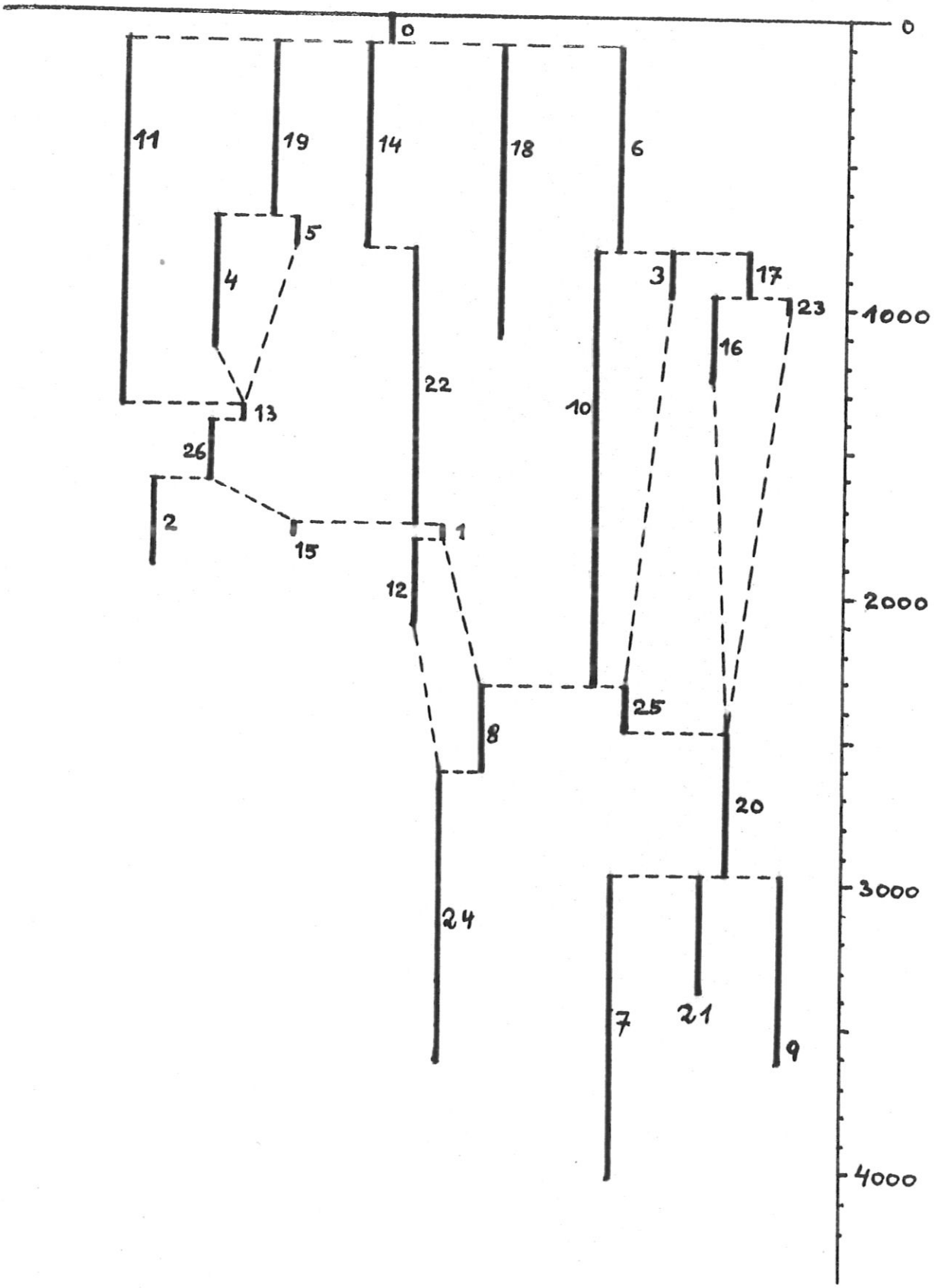


Fig. 3 Storage chart for the program of Fig.2, evaluated by the described procedure.