MODELING, IDENTIFICATION AND CONTROL, 1990, VOL. 11, NO. 1, 13–41
doi:10.4173/mic.1990.1.2

An improved heuristic algorithm for selection of tear streams and precedence ordering in process flowsheeting computations

KRISTIAN M. LIEN† and TERJE HERTZBERG‡

Keywords: *Tearing, decomposition, precedence ordering, process flowsheeting.*

This paper presents a new algorithm based on the heuristic tearing algorithm by Gundersen and Hertzberg (1983). The basic idea in both the original and the proposed algorithm is sequential tearing of strong components which have been identified by an algorithm proposed by Targan (1972). The new algorithm has two alternative options for selection of tear streams, and alternative precedence orderings may be generated for the selected set of tear streams. The algorithm has been tested on several problems. It has identified minimal (optimal) tear sets for all of them, including the four problems presented in Gundersen and Hertzberg (1983) where the original algorithm could not find a minimal tear set. A Lisp implementation of the algorithm is described, and example problems are presented.

1. Introduction

The leading simulators on the commercial market for process flowsheeting systems (Aspen Technology 1986, Simulation Sciences Inc., 1988) are still based on a sequential modular approach. This means that each unit is calculated in turn. Thus, to be able to calculate the output of a process unit, its input must be known. If there are information loops in the flowsheet to be simulated, then the input to one or more units must be guessed, and the calculation must be solved iteratively.

If the flowsheet contains many loops it may be difficult to converge the iterative calculations. Divergence is frequently observed in flowsheeting calculations where there are mass or energy loops or design specifications (the output of the unit is specified instead of the input). In this context the selection of which variables to guess and iterate—the tear variables, as well as how the tear variables are updated, are important issues. The overall goal is naturally to devise a scheme that is such that optimal convergence properties can be guaranteed. This is, however, still an unsolved problem. Which variables to select as iteration variables has usually been approached as a separate problem, although it is clear that convergence properties are closely coupled to numerical properties of both the process unit models and to the update scheme used in the iterative calculations. The present paper does also take such a separate approach. Only the structure of the flowsheet is considered, numerical properties are not investigated.

1.1. *Tearing*

Westerberg *et al.* (1979) and Gundersen and Hertzberg (1983) provide a solid background to the tearing problem and the different methods of solution proposed in

Received 1 August 1989.
† Laboratory of Chemical Engineering, The Norwegian Institute of Technology.
‡ SINTEF, Division of Applied Chemistry, N-7034 Trondheim, Norway.

the literature. These two references may be consulted for further references. It should be pointed out, however, that a major part of the proposed methods have adopted variants of a common heuristic: the number of tear variables generated is taken as a measure of an algorithm's performance. Frequently used variants of this approach to the problem are minimization of the number of tear streams, or the minimization of the weighted number of streams, where individual weights are assigned to each stream and meant to reflect the desirability of having the stream as a tear stream. The purpose of the latter variant is naturally to use the weights in such a manner that extratopological considerations may be accounted for (e.g. the size of the stream, the degree of nonlinearity of the process units it connects, etc.).

In spite of these simplifications, the remaining part of the problem is still hard: Karp (1972) has stated that algorithms that guarantee global optimality, like fewest number of tear streams, will be NP-complete. This is not a severe restriction for many industrial size problems, however. Algorithms that guarantee a minimum number of tear streams for reasonable size problems have indeed been developed, as pointed out by Gundersen and Hertzberg (1983). It should be realized, though, in a process flowsheeting context, that a minimization of the number of tear streams is merely a heuristic. A minimum number of tear streams does not necessarily guarantee the best convergence properties, although it is generally recognized that fewer is usually better than more.

The present paper proposes improvements to an algorithm presented by Gundersen and Hertzberg (1983) and Gundersen (1982). The Gundersen algorithm is based on local criteria, and its time complexity is empirically found to be slightly faster than $O(n^2)$. It cannot guarantee a minimum number of tear streams, but most of the time it comes very close.

1.2. *Precedence ordering*

Before a flowsheet is torn it is structurally a graph that may involve cycles. Tearing removes some of the arcs in the graph so that it becomes acyclic, and at least one of the nodes in the acyclic graph will have no arcs leading to it. This node is termed the root node. In case there is more than one such node, an artificial root node may be constructed with all nodes without incoming arcs as immediate successors. Thus there will always be one uniquely defined root node in the graph.

A precedence ordering of the units in a flowsheet may be defined as a traversal of the acyclic graph representing the torn flowsheet from the root node. Generally there will be more than one traversal of an acyclic graph. This is emphasized because it does not seem to be recognized: one usually expects to find alternative precedence orderings for any selected set of tear streams in a process flowsheet. If loops are nested within loops in the original untorn flowsheets and it is decided to converge these loops one at a time, then two different alternative precedence orderings may give rise to one computational sequence that converges substantially faster than the other, without any modification of the tear set.

2. Description of the Gundersen algorithm

In the Gundersen algorithm the process flowsheet is represented as a directed graph with weights associated with each arc. The goal is to make the graph acyclic by the removal of arcs such that the weighted number of arcs removed is minimal. Arcs are removed sequentially, and the decision about which arcs to remove next is based on

local information. The algorithm terminates when there are no more cycles in the graph. It is based on repeated application of two steps:

(1) Find all strong components in the graph, where a strong component is defined as a maximal set of nodes S with the property that any node in S is reachable from any other node in S following a path along zero or more arcs.

(2) For any strong component found in (1), if its cardinality is greater than one (i.e. if it is not a single node) then one of its nodes is selected as a tear node, meaning that all input arcs to it are torn. The tear node selection criterion employed in the algorithm is: Select the node with maximum ratio of weighted output cardinality to weighted input cardinality in the strong component. Formally this may be stated as:

$$\text{Tearnode}_{\text{comp}_i} = \left\{ j \middle| \operatorname*{Max}_{j \in \text{comp}_i} \left\{ \sum_{k=1}^{oc_j} w_{jk} \Big/ \sum_{l=1}^{ic_j} w_{lj} \right\} \quad (k, l \in \text{comp}_i) \right\} \tag{1}$$

with

comp_i the set of nodes comprising strong component i
oc_j the output cardinality of node j
ic_j the input cardinality of node j
w_{jk} weight on arc from node j to node k
w_{lj} weight on arc from node l to node j

The well known algorithm by Tarjan (1972) is used for the detection of the strong components. When all the input arcs to a node in a strong component have been removed the reduced graph may still be cyclic, so the above two steps have to be repeated until Tarjan's algorithm can only find strong components with a cardinality equal to one. The following metaprogram describes this procedure:

```
Begin
  Precedence-ordering:=NIL;
  Strong-components:=Result of application of Tarjan's algorithm to the original
  graph;
  TEARGRAPH (Strong-components);
End;

Procedure TEARGRAPH (Strong-components)
begin
  For C ∈ Strong-components do
    If Cardinality (C)=1
   Then
      If NOT C ∈ Precedence-ordering
      Then
        Precedence-ordering:=Append (Precedence-ordering, C)
      Endif
    Else
    begin
      Select Tear-node ∈ C according to Eq. (1) above;
      Remove all input arcs to Tear-node;
      New-strong-components:=Result of application of Tarjan's algorithm to the
      reduced graph;
```

```
      TEARGRAPH (New-strong-components);
    Endif
  Endfor
End procedure TEARGRAPH;
```

The actual implementation of the algorithm presented in Gundersen and Hertzberg (1983) is iterative rather than recursive. For further details the reader is referred to the reference, where the Pascal implementation is also listed.

3. The new algorithm

3.1. *Tearing*

A strong component S in a graph G is defined as a maximal set of nodes in G with the property that all nodes in S are reachable from any other node in S by following zero or more streams. A single node is a strong component with cardinality 1.

Any strong component S can be decomposed into one or more strong components s_i by tearing streams between nodes in S. These new strong components can be single nodes, or they can be sets of strongly connected nodes. From this it is evident that the strong components s_i generated by tearing S are disjoint subsets of S. To illustrate this, suppose the contrary, that there may be a node X which is contained in more than one of the components $s_1 \ldots s_n$. Then all nodes in s_1 will be reachable from X, all nodes in s_2 will be reachable from X, etc., and X will be reachable from all nodes in s_1, all nodes in s_2, etc. This means, from the definition of strong components, that the union of nodes present in any of the strong components that X is a member of comprises one strong component, which is a contradiction of the assumption that S contains two strong components.

This has the following consequences. When tearing the strong components in the graph, single nodes can be blocked from the remaining problem as soon as they are encountered. Single nodes can never be contained in any strong component as yet undiscovered. If a single node which has already been encountered could be contained in a strong component as yet undiscovered, it would have to be an element of two disjoint sets. Further, a node which has been selected as a tear node can also immediately be blocked from the remaining problem. When node X has been selected as tear node in the strong component S, either all the streams to it from other nodes in S, or (in the new algorithm) all the streams from it to other nodes in S, are torn. This implies that in the remaining problem there is either no way of reaching X from any other node in S or there is no way of reaching any other node in S from X. This implies that X will be a single node in the remaining problem, and thus it can be blocked immediately. Blocking here means that a node is not to be visited during the remaining part of the search for tear streams.

These findings can be used to improve the efficiency of the original algorithm. Using Tarjan's algorithm iteratively, parallel tearing of all the strong components detected in one iteration is suggested. On a sequential machine 'parallel' should be read as 'independently, sequentially'. This will reduce the number of invocations of Tarjan's algorithm significantly. The tearing process can be considered like growing a tree. The root node is the entire initial graph, the leaf nodes are all the single nodes and the interior nodes of the tree are strong components with cardinality greater than 1. In the original algorithm the number of invocations of Tarjan's algorithm will be equal to one plus the number of interior nodes in the tree. Parallel tearing of strong components will

reduce the number of invocations of Tarjan's algorithm to a number equal to the height of the tree.

The finding that there are nodes which may be excluded early from the search for tear streams, may be used to reduce the work done inside Tarjan's algorithm. Tarjan's algorithm searches the graph for strong components. Since we are only interested in components which are not yet discovered, and identified single nodes cannot be contained in these, Tarjan's algorithm does not need to be applied to single nodes which are already found.

In addition to the original criterion for tear node selection, a new criterion has been designed and implemented. Either of these criteria may be used in the new algorithm.

The new criterion

In a strong component S, select the tear node as the node with

$$\text{Tearnode}_{\text{comp}_i} = \left\{ j \middle| \underset{j \in \text{comp}_i}{\text{Max}} \left\{ \text{loops}_j \middle/ \sum_{k=1}^{oc_j} w_{jk},\ \text{loops}_j \middle/ \sum_{l=1}^{ic_j} w_{lj} \right\} \quad (k, l \in \text{comp}_i) \right\} \tag{2}$$

with

comp_i	the set of nodes comprising strong component i
oc_j	the output cardinality of node j
ic_j	the input cardinality of node j
w_{jk}	weight on arc from node j to node k
w_{lj}	weight on arc from node l to node j
loops_j	the number of unit cycles in which node j is contained

If the weighted output cardinality of the tear node is greater than its weighted input cardinality then all the input arcs to the node are torn. Otherwise all the output arcs from it are torn.

Since the unit cycles are used in the selection of tear nodes, this information must be generated, stored, and updated when streams are torn. Another algorithm by Tarjan (1973) is used to identify all the unit cycles.

3.2. *Precedence ordering*

For any given tear set there is usually more than one precedence ordering. The original algorithm generates only one of these. The new algorithm is capable of finding a specified number of these by employing a simple backtracking strategy.

Generally, a valid precedence ordering P on a graph G is a sequence of nodes where every node in P either:

(i) has no predecessor in G, or
(ii) has every predecessor in G preceding it in P.

The precedence ordering is to be calculated on the torn graph. As a result, streams from the original graph will be absent during the precedence ordering. Unless proper care is taken of this, it may result in precedence orderings where for example a part of one strong component is calculated, then a part of another strong component, then some simple nodes, before the first strong component is completed. This kind of behaviour must be avoided; additional constraints must be imposed on the precedence ordering.

The general criterion for valid precedence ordering has therefore been augmented by the following logic. The ordered pair of nodes (A, B) may be part of a precedence ordering P if

A and B are contained in the same strong component

or

A is contained in a strong component $S1$ and B is contained in another strong component $S2$ and all other nodes in $S1$ precedes A in P and all other nodes outside $S2$ which have streams leading to any node in $S1$ precedes A in P.

Precedence ordering which does not obey this logic may include nodes in iterative calculations that should have been left outside the iterative loop.

Alternative means of precedence orderings

A partial precedence ordering is a partially completed sequence of nodes obeying the restricted precedence criterion above. Alternative means of precedence ordering occur whenever a partial precedence ordering of length i is computed and there is more than one node available as node $i+1$ in the sequence. Every valid extension of the sequence will be a valid partial precedence ordering of length $i+1$. By keeping track of how many alternative partial sequences we already have found, we are also able to stop the search for alternatives if we exceed a user specified number of alternatives, and can from then on simply complete all the partial sequences that have been found.

4. Examples and results

Before we proceed to a description of the new algorithm, examples of precedence ordering and the selection of tear streams are presented.

4.1. *Precedence ordering*

Figure 1 (*a*) depicts a graph consisting of four strong components; {1 2 3 7 8 9 10}, {4}, {5} and {6}. Two streams must be removed to decompose the first of these into strong components with cardinality one. The new algorithm, using criterion (2), suggests that streams 7→1 and 10→1 are removed. The torn graph, depicted in

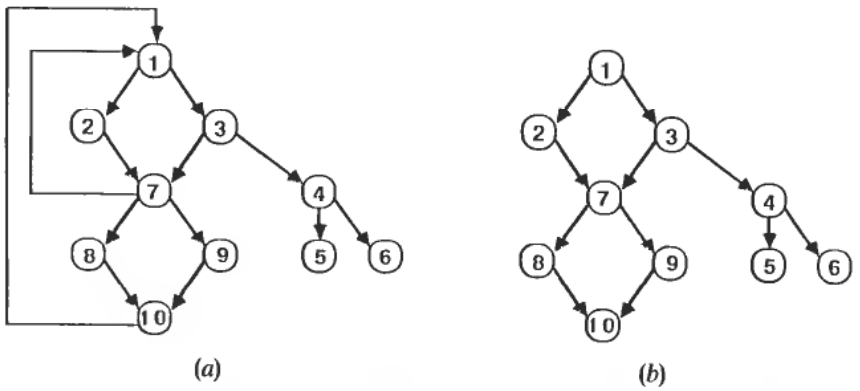


Figure 1. An example of precedence ordering. (*a*) The original graph, (*b*) the graph after tearing 7→1 and 10→1.

Fig. 1 (*b*) has node 1 as the root node, and there is clearly more than one way to traverse the graph. The new algorithm found the following eight

[1 2 3 7 8 9 10 4 5 6]
[1 2 3 7 8 9 10 4 6 5]
[1 2 3 7 9 8 10 4 5 6]
[1 2 3 7 9 8 10 4 6 5]
[1 3 2 7 8 9 10 4 5 6]
[1 3 2 7 8 9 10 4 6 5]
[1 3 2 7 9 8 10 4 5 6]
[1 3 2 7 9 8 10 4 6 5]

These are not the only possible ones. Other means of ordering exist, but these are inferior to the above eight. Examples of such inferior orderings

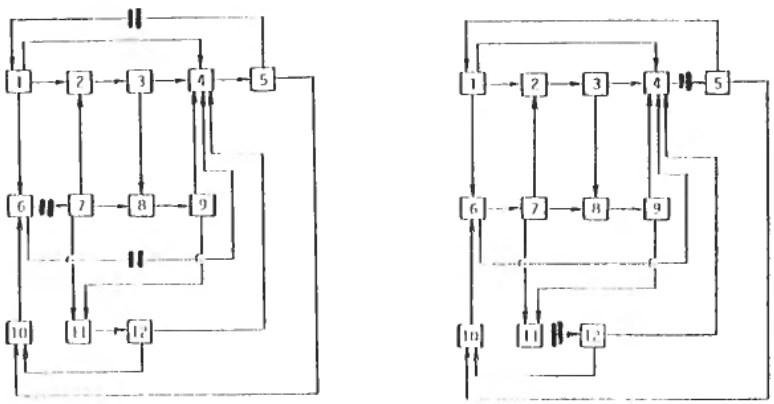
[1 2 3 4 5 6 7 8 9 10]
[1 2 3 4 7 8 9 10 5 6]
[1 3 2 4 6 5 7 8 9 10] etc.

Why are these orderings considered to be inferior? Clearly nodes 4, 5 and 6 have no effect on iterative calculations on the nodes involved in the first strong component; {1 2 3 7 8 9 10}. On the other hand, all nodes in this component have an effect on nodes 4, 5 and 6. Therefore no computations involving nodes 4, 5 and 6 should be performed until computations involving any of the nodes in the first strong component are converged.

4.2. *Selection of tear streams*

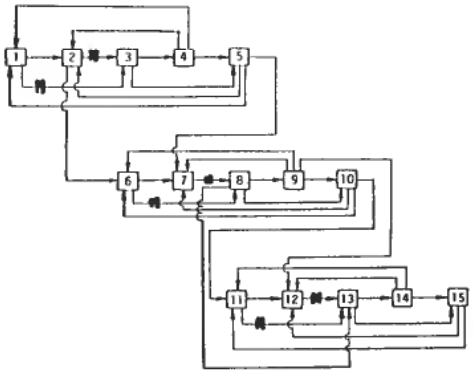
The four examples presented here are the same as the four examples in Gundersen and Hertzberg (1983) where the original algorithm failed to find a minimal tear set. The three first examples are originally from Pho and Lapidus (1973), Barkley and Motard (1972) and Christensen and Rudd (1969). Example 4 is the heavy water plant proposed a benchmark test in Gundersen and Hertzberg (1983). Figures 2–5 depict the graphs with selected tear streams marked both for the original and the new algorithm, with the results from the former in the (*a*) figures and from the latter in the (*b*) figures. The results are also summarized in Table 1.

The new algorithm finds minimal tear sets in the four examples. In example 4, criterion (2) is not practical because of the extreme complexity of the graph. The 109 nodes graph has more than thirteen thousand elementary cycles, and in order to use criterion 2 all these must be stored. The criterion used is the same as in the original Gundersen algorithm, and using the same tear node selection criterion, one would expect the same tear nodes to be selected. This obviously does not happen, because when the criterion gives its best value to two or more nodes, the choice between these nodes is arbitrary in both algorithms (actually, the first node encountered with this best value is chosen). Because the two algorithms do the search in different ways, the first node encountered with the best value may not be the same in the two algorithms. However, the nodes will have the same value in both algorithms when the same criterion is used, it is only the order in which they are visited during the search that differs. This may work favourably for the original implementation in some cases and for the new one in others, and it is impossible to tell which of them it will be in advance.

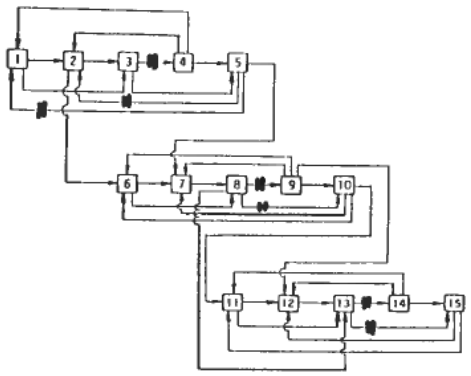


(a) (b)

Figure 2. Problem 1. (*a*) Original algorithm, (*b*) new algorithm.



(*a*)



(*b*)

Figure 3. Problem 2. (*a*) Original algorithm, (*b*) New algorithm.

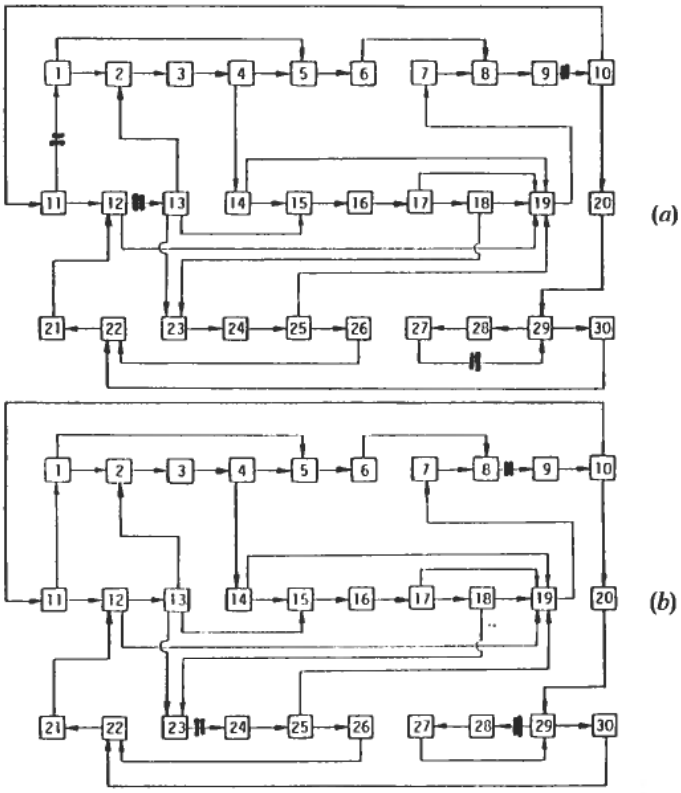
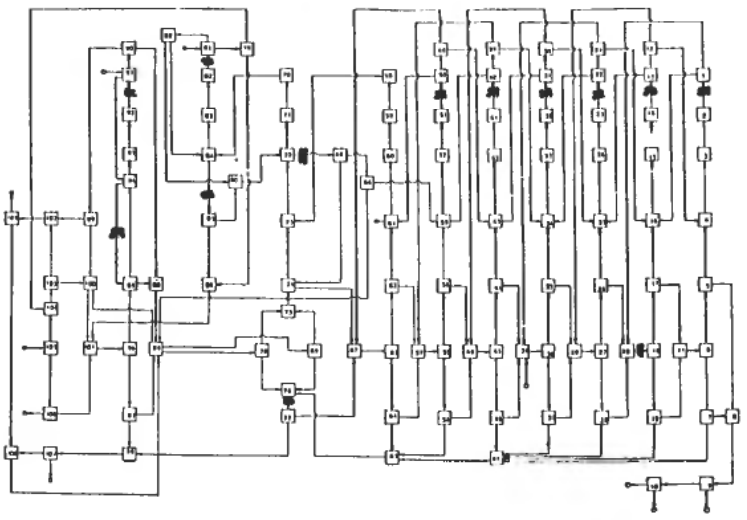


Figure 4. Problem 3. (*a*) Original algorithm, (*b*) New algorithm.

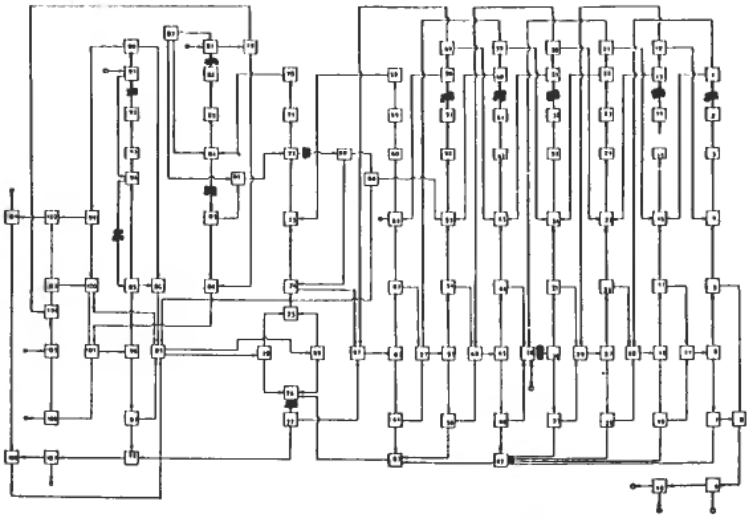
5. The new algorithm: description of the implementation

The following metaprogram describes the overall structure of the new algorithm.

```
Procedure teargraph (Input-data, desired-number-of-precedence-orderings,
                     tearnode-selection-criterion)
begin
  Build necessary internal datastructures from input-data;
  If the new tearnode-selection-criterion is being used
    Then find and store all unit-cycles in the graph;
  Find the strong components in the graph;
    With cardinality greater than one
  While strong components remain do
  begin
    For all comp ∈ strong components do
      begin
        Select a tearnode in comp;
        Tear the graph by tearing comp according to the tear node selected;
      end;
    endfor;
  If the new tearnode-selection-criterion is being used
    Then update the unit-cycle information;
  Find the strong components in the remaining graph;
  endwhile;
  generate desired-number-of-precedence-orderings;
end procedure teargraph.
```



(*a*)



(*b*)

Figure 5. Problem 4. (*a*) Original algorithm, (*b*) New algorithm.

Table 1. Tear stream selection summary.

Example problem	Problem no. in Gundersen and Hertzberg (1983)	Number of tear streams		Criterion used in the new algorithm
		Original algorithm	New algorithm	
No. 1	No. 2	3	2	(2)
No. 2	No. 3	7	6	(2)
No. 3	No. 7	4	3	(2)
No. 4	No. 10	13	12	(1)

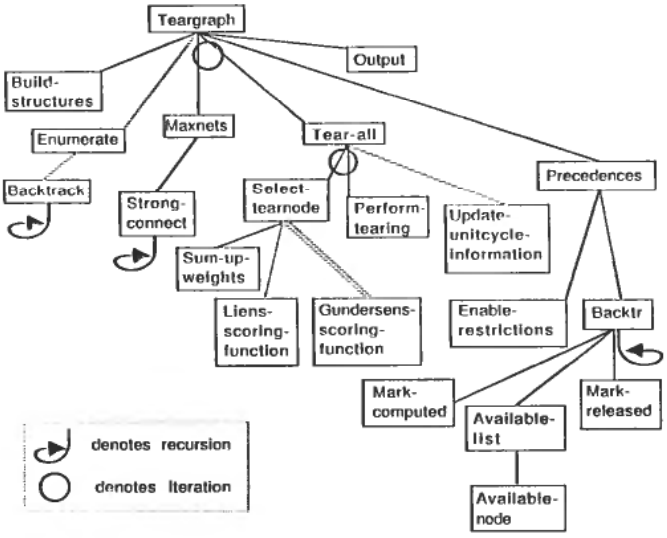


Figure 6. The structure of the Lisp implementation of the new algorithm.

5.1. *The major functions*

The algorithm is implemented as a set of Lisp functions in Golden Common Lisp, Gold Hill Computers (1985), a subset of the Common Lisp language (Steele (1984) for personal computers. Figure 6 depicts the calling hierarchy of the major functions comprising the Lisp implementation of the algorithm. The functionality of these functions is briefly described below:

Teargraph: top level function

Build-structures: transform the input data to the major part of the datastructures used by the algorithm.

Enumerate: detects unit cycles by iterative use of function backtrack. Does also do bookeeping on global unit cycle variables.

Backtrack: backtracking function (recursive) to identify unit cycles.

Maxnets: detects strong components by iterative use of function strong-connect.

Strong-connect: recursive function which identifies strong components and blocks single node strong components.

Tear-all: selects a tearnode in every strong component returned by maxnets and tears it. If the new tearnode selection criterion is being used, the unit cycle information is updated.

Select-tearnode: Finds the tearnode with the highest score in a strong component using the functions sum-up-weights and one of Liens-scoring-function of Gundersens-scoring-function on every node in the strong component.

Sum-up-weights: the in-weight and out-weight at all the nodes in a strong component are calculated. (The in-weight of a node in a component is the sum of the weights of all streams leading to the node from other nodes in the component).

Liens-scoring-function: Calculates the score of a node as potential tearnode using the new selection criterion, and determines whether the streams leading to it or from it should be torn.

Gundersens-scoring-function: Calculates the score of a node as a potential tearnode using the old tearnode selection criterion. Using this criterion, only input streams to the node may be torn.

Perform-tearing: Tears a strong component after select tearnode has detected the tearnode. Does also block the tear nodes.

Update-unitcycle-information: Torn unit loops are deleted from the global cycle list, and all nodes occurring in these have their loops bookkeeping association altered.

Precedences: Initializes datastructures enabling restrictions in the precedence-orderings to be made using function enable-restrictions and generates a given number of precedence orderings using the backtracking function backtr.

Enable-restrictions: Generates datastructures that keep records of the sets of nodes that are members of any particular strong component, and also of streams outside any component that lead into it.

Mark-computed: A node is marked computed when it is in a current partial precedence ordering.

Backtr: Recursive function which finds all alternative available next modes in a sequence using function available-list and itself in lower recursive levels.

Mark-released: When the current precedence ordering is complete, the nodes in it are released so they may be contained in other alternative precedence orderings.

Available-list: Finds all alternative next nodes which are available in a sequence using function available-node. Adjacent nodes are considered before nodes further away (to prevent 'jumping around in the graph').

Available-nodes: A node is available as a next node in a sequence if all its predecessors in the graph are marked computed and it does not cause a premature jump out of a strong component or a premature jump into a strong component. (Preventing these jumps are restrictions mentioned in the description of functions precedence and enable-restrictions.)

5.2. *Input-representation of the graph*

A list 'data' of lists 'adjacent$_i$' of 2-element-lists 'adj-to$_j$' consisting of numbers 'destination$_j$' and 'weight$_{ij}$', with the interpretation that node no. i in the graph has streams leading to other nodes in the graph numbered 'destination$_j$' and that the weight of the streams are 'weight$_{ij}$'.

```
Example data = (((1  1)(2  2))
                ((2  1)(3  1))
                ((1  1))
                (0))
```

This is the input representation of the graph shown in Fig. 7, with weight = 1 for all streams except the stream between 0 and 2. This stream has weight = 2. Note the convention that the first node is numbered 0 and not 1.

5.3. *Major datastructures*

For efficiency, the graph is represented as a vector of adjacency-lists, rather than a list of adjacency-lists. Each element of the vector (called ADJACENT) is a list of this node's immediate successors. In addition, a similar vector of immediate predecessors is used, called PREDECESSORS. The weights are efficiently accessible by making a

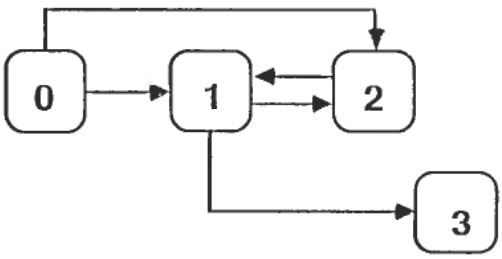


Figure 7. Input representation example.

vector from the input-representation of the graph, where each element i of the vector will be interpreted as an association list with keys 'destination$_j$' and values 'weight$_{ij}$'.

Vectors INS, OUTS and LOOPS are used so that the tearnode selection criterion functions can be executed efficiently. These are vectors of numbers with the interpretations: INS contains numbers describing how many streams that enter the nodes from other nodes in a same strong component at a given time during the tearing process. OUTS, similar for leaving streams. LOOPS describes how many remaining unit cycles a node is contained in at a given time.

NUMBER and OUTS are vectors which have different interpretations in different parts of the algorithm (see description of the functions where they occur).

LOWLINK, MARK and ADJACENT-X are locally used vectors.

NUMBER-OF-NODES occurs globally in most functions and is a number; the highest numbered node in the graph.

CYCLE-LIST is a global variable in most functions where it occurs and is the list of all remaining unit-cycles at any time.

TEAR-SET occurs globally and is the list of all tear streams (each tear stream is represented as a 2-element list; a from-node and a to-node).

All vectors are accessed by means of automatically defined access functions. When a vector is declared (using function MAKEVECTOR), the vector is defined, and at the same time two functions are generated. One access function for accessing the values stored in the vector and one function for altering the values. Example:

(make-vector 'NUMBER' (1 2 3 4 5))

The vector is established, a function named NUMBER is generated to access the elements of the vector, and a function named NUMBER0 is generated to alter the contents of the vector when needed.

6. Conclusions

The new algorithm is a modification and extension of the original Gundersen algorithm Gundersen and Hertzberg (1983). It has two alternative criteria for tearnode selection and a backtracking procedure for the generation of alternative precedence orderings.

In all the four example problems reported, the original implementation of the Gundersen algorithm found no minimal tear set. The new tear node selection criterion gave a smaller tear set than the original criterion in three of the four example problems. In one of the example problems, the new implementation of the original criterion gave a minimal tear set.

The new tearnode selection criterion is more demanding both in space and time than the original, because the cycle information has to be generated, stored and updated. Thus this criterion is only practical for graphs with a limited number of elementary cycles. Example number four, which has more than thirteen thousand elementary cycles, was too large to be solved on a 640 K personal computer using the new tearnode selection criterion. In this example the new implementation of the original criterion surprisingly found a minimal tear set.

The new algorithm clearly demonstrates that any selected set of tear streams may have more than one precedence ordering, and it is able to identify all such orderings. In cases with an unpractical number of such orderings, the user may specify an upper limit on the number of orderings that should be generated and displayed. Inferior precedence ordering which include nodes inside iterative calculations, that do not belong to the loop in question, are not generated by the algorithm.

Acknowledgments

The authors are grateful to the Royal Norwegian Council for Scientific and Industrial Research (NTNF), Statoil and Norsk Hydro for partially funding this research.

References

Aspen Technology (1986). ASPEN PLUS *Introductory Manual*, Cambridge, Massachusetts.
Barkley, R. W., and Motard, R. L. (1972). *Chem. Eng. J.*, **3**, 265.
Christensen, J. H., and Rudd, D. F. (1969). *AIChE. J.*, **15**, 94.
Gold Hill Computers (1985). *Golden Common Lisp*, Cambridge, Massachusetts.
Gundersen, T. (1982). Decomposition of Large Scale Chemical Engineering Systems. Ph.D. thesis, The Norwegian Institute of Technology, Trondheim, Norway.
Gundersen, T., and Hertzberg, T. (1983). *Modeling, Identification and Control*, **4**, 139.
Karp, R. M. (1972). *Complexity of Computer Computations* (Plenum Press, New York), p. 85.
Pho, K., and Lapidus, L. (1973). *AIChE J.*, **19**, 1170.
Simulation Sciences, Inc. (1982). *PROCESS Reference Manual*, Fullerton, California.
Steele, G. L. (1984). *Common Lisp*. Digital Press. Billerica, Massachusetts.
Tarjan, R. (1972). *SIAM J. Comp.*, **1**, 146; (1973). *Ibid.*, **2**, 211.
Westerberg, A. W., Hutchison, H. P., Motard, R. L., and Winter, P. (1979). *Process Flowsheeting*, Cambridge University Press.

Appendix A: Documented Lisp code

```
;*************************************************************************
;
; LISP program for near–optimal tearing and generation of alternative
; precedence orderings.
;                                Kristian Lien,
;                                Laboratory of Chemical Engineering,NTH,
;                                August 1989
;*************************************************************************

; CONVENIENT VECTOR–MANIPULATION (automatic def. of access–functions) :


(defun build–vector (name vector)
  `(and
   (setf ,name ,(append '(vector) (q–v vector)))
   (defun ,name (index) (aref ,name index))
   (defun ,(makename name) (index new–value)
     (setf (aref ,name index) new–value))))

;        Builds a data–structure which is later evaluated (executed)
;        by function make–vector.
;        The above function is the logical equivalent of the first pass
;        of an ordinary LISP–macro.


(defun q–v (list)
  (mapcar '(lambda (x) (cond ((listp x) (cons 'quote(list x))) (t x))) list))

;        This function prevents the elements of a list from being
;        interpreted as functions if the elements are list:
;        If an element <x> is a list, the element is transformed to
;        '<x>


(defun makename (name)
  (gensym (string name))
  (intern (string (gensym 0))))

;        Symbol–generation is very restricted in Golden Common Lisp
;        The awkwardness of the above function is due to this.
;        Receiving the symbol <name> as actual parameter,
;        the function generates and interns the symbol <name>0
```

```
(defun make–vector (name vector)
  (eval(build–vector name vector)))
;          Given a symbol <name> and a list <vector> of length x,
;          function make–vector creates a vector named <name> with x
;          elements.
;          An access–function named <name> is also generated, and a
;          function named <name>0 for altering the contents of the
;          vector's elements.
;===============================================================

; TOP–LEVEL AND OUTPUT–FUNCTION :


(defun teargraph (data tear–out prec–number method)
  (setq *print–length* nil)
  (setq *print–level* nil)
  (prog (scoring–function strong–components tear–set cycle–list strongcomps
       loops ins outs lowlink adjacent adjacent–x predecessors )
      (build–structures data)
      (cond ((equal method 'lien)
            (enumerate)
            (setq scoring–function 'liens–scoring–function))
           (t (setq scoring–function 'gundersens–scoring–function)))
      (setq strongcomps (maxnets))
      (setq strong–components strongcomps)
   loop
      (cond ((null strong–components) (go endloop))
           (t (tear–all strong–components scoring–function)
             (setq strong–components (maxnets))
             (go loop)))
   endloop
      (output tear–set tear–out
            (precedences strongcomps prec–number)
            prec–number))
  (format t "~% ~% Have a nice day")
  'bye)


(defun output (tear–set tear–out precedence prec–number)
  (cond (tear–out (format t "~% ~% ~% Tear–streams : ~% ~S " tear–set)))
  (format t "~% ~% Sequence(s)  :")
  (mapcar '(lambda (x) (format t "~% ~S" x)) precedence)
  (cond ( (> prec–number (length precedence))
        (format t "~% Only ~S sequences were found" (length precedence)))))
```

```
;=============================================================

; BUILDING OF INTERNAL DATASTRUCTURES FROM INPUT-DATA


(defun build-structures (data)
  (setq number-of-nodes (- (length data) 1))
  (let ((ad-data (mapcar '(lambda (x)
       (cond ((null x) nil)
            (t (mapcar '(lambda (y) (cond ((null (car y)) nil)
                                          (t (car y))))
                       x))))
                  data)))
  (make-vector 'weights data)
  (make-vector 'adjacent  ad-data)
  (make-vector 'adjacent-x ad-data)  ; destroyed in "enumerate"
  (make-vector 'predecessors (make-list (+ 1 number-of-nodes)))
  (make-vector 'number (make-list (+ 1 number-of-nodes)
                                  :initial-element -1))
  (make-vector 'lowlink(make-list (+ 1 number-of-nodes)
                                  :initial-element -1))
  (make-vector 'ins  (make-list (+ 1 number-of-nodes)
                                :initial-element 0))
  (make-vector 'outs (make-list (+ 1 number-of-nodes)
                                :initial-element 0))
  (make-predecessors )))


(defun make-predecessors ()
  (do ((j 0 (+ 1 j)))
      ((> j number-of-nodes))
      (cond ((adjacent j)
            (dolist (adj (adjacent j) nil)
                 (predecessors0 adj (cons j (predecessors adj))))))))


; Access-function for "weights" :

(defun weight (from to)
  (cadr (assoc to (weights from))))

;=============================================================
```

```
; FUNCTIONS FOR IDENTIFICATION OF STRONG COMPONENTS IN THE GRAPH


(defun maxnets ()
  (do (( j 0 (+ 1 j)))
      (( > j number–of–nodes))
      (cond ((neq (number j) 'blocked)
             (number0 j –1)
             (lowlink0 j –1))))
  (let ((i –1) (stack nil) (strongcomps nil))
       (do ((j 0 (+ 1 j)))
           (( > j number–of–nodes) strongcomps)
           (cond ((eq (number j) –1)
                  (strongconnect j))))))

; Description: For j from 0 to number–of–nodes do
;                If node j is not already blocked then
;                  number(j) := –1;
;                  lowlink(j) := –1;
;                Endif;
;              Endfor;
;              i := –1; stack := NIL; strongcomps := NIL;
;              For j from 0 to number–of–nodes do
;                If number(j) = –1
;                   (i.e. if node j is yet unvisited AND not blocked) then
;                  strongconnect(j);
;                Endif;
;              Endfor;
;              Result strongcomps;
; End description

(defun strongconnect (v)
  (prog (a)
     (setq i (+ 1 i))
     (lowlink0 v i)
     (number0 v i)
     (push v stack)
     (mapcar '(lambda (w)
                (cond ((eq (number w) –1)
                       (strongconnect w)
                       (lowlink0 v (min (lowlink w) (lowlink v))))
                      ((and (member w stack) ( < (number w) (number v)))
                       (lowlink0 v (min (lowlink w) (lowlink v))))))
             (adjacent v))
     (cond (( = (number v) (lowlink v))
            (loop
              (cond ((and stack ( >= (number (car stack)) (number v)))
                     (push (pop stack) a))
                    (t (return))))
            (cond ((eq (length a) 1)
                   (number0 (car a) 'blocked))
                  (t (push a strongcomps)))))))
```

```
; Description: i := i + 1; number(v) := i; lowlink(v) := i;
;              Push node v on the stack stack;
;              For all nodes w adjacent to node v in the graph do
;                If node w is yet unvisited AND not blocked then
;                  strongconnect(w); (i.e. recursion)
;                  lowlink(v) := min { lowlink(v), lowlink(w) } ;
;                Elseif node w is on the stack stack
;                      AND number(w) < number (v) then
;                  lowlink(v) := min { lowlink(v), lowlink(w) };
;                Endif;
;              Endfor;
;              If number(v) = lowlink(v)
;                (which is the case for the first encountered nodes of
;                 strong components AND single nodes ONLY)
;              then a:= the nodes on the stack stack from the top down to
;                       and including node v;
;                   Remove the nodes in a from the stack;
;                   If only one node in a then block this node (which is v)
;                   Else push a onto strongcomps
;                        (a will be a new strong component)
;                   Endif;
;              Endif;
; End description
;=============================================================

; Tarjan's unit–cycle enumeration algorithm
;   (Reference : SIAM J. COMPUT. Vol.2 No.3,page 211 (Sept.1973))

;   (Enumerate and backtrack is an exact implementation of the
;    algorithm described in the above referenced paper)

(defun enumerate ()
  (let (mark pointstack markedstack )
   (make–vector 'loops (make–list (+ 1 number–of–nodes) :initial–element 0))
   (make–vector 'mark (make–list (+ 1 number–of–nodes)))
   (setq cycle–list nil)
      (do ((rootnode 0 (+ 1 rootnode)))
          ((> rootnode number–of–nodes))
          (backtrack rootnode rootnode)
          (dolist (dummy markedstack nil)
                  (mark0 (pop markedstack) nil)))
      (dolist (cycle cycle–list nil)
              (dolist (node cycle nil)
                      (loops0 node (+ 1 (loops node)))))))
```

```
(defun backtrack (v rootnode)
  (prog (f)
    (setq f nil)
    (mark0 v t)
    (push v markedstack)
    (push v pointstack)
    (dolist (w (adjacent–x v) nil)
      (if w
        (cond ((< w rootnode) (adjacent–x0 v (delete w (adjacent–x v))))
              ((= w rootnode)
               (setq f t)
               (push (member rootnode (reverse pointstack)) cycle–list))
              ((not (mark w))
               (setq f (or (backtrack w rootnode) f))))))
    (cond (f (prog (x)
                loop (setq x (pop markedstack))
                     (mark0 x nil)
                     (cond ((not (= v x)) (go loop))
                           (t (return))))))
    (pop pointstack)
    (return f)))

;           Enumerate and backtrack store information about the
;           unit–cycles and the nodes they contain in the list
;           cycle–list and the vector loops.
;           Cycle–list contains all identified unit–cycles.
;           Element j in the loops vector contains information about
;           how many unit–cycles node j is contained in.
;
;           During the tearing process unit–cycles are being torn.
;           See function update–unitcycle–information for description
;           of how this is handled.
;=============================================================

; FUNCTIONS FOR SELECTION OF TEAR–NODES,
;            PERFORMING THE TEARS
;         AND UPDATING OF CYCLE–INFORMATION :

(defun tear–all (strong–components scoring–function)
  (let (tearnodes type–and–node tear–type tear–node)
    (dolist (comp strong–components nil)
          (setq type–and–node
            (select–tearnode comp scoring–function))
          (setq tear–type (first type–and–node))
          (setq tear–node (second type–and–node))
          (push tear–node tearnodes)
          (perform–tearing tear–type tear–node comp))
    (cond ((equal scoring–function 'liens–scoring–function)
          (update–unitcycle–information tearnodes)))))
```

```
; Description: For every comp in strong–components do
;              Find a tear–node in comp and decide whether the streams
;              to it or from it should be torn using function
;              select–tearnode;
;              Perform the tearing of comp using function
;              perform–tearing;
;            Endfor;
;            If the new tearnode–selection–criterion is being used then
;              perform book–keeping on the stored unit–cycle information
;              using function update–unitcycle–information;
;            Endif;
; End description


(defun select–tearnode (comp scoring–function)
  (let (best (best–score 0) x)
    (dolist (node comp nil)
      (outs0 node 0)
      (ins0  node 0))
    (sum–up–weights comp)
    (dolist (node comp best)
      (setq x (funcall scoring–function node))
      (cond (( > (cadr x) best–score)
            (setq best–score (cadr x))
            (setq best (list (car x) node)))))))

; Description: For every node node in the strong component comp do
;              Set the initial value of the weighted number of streams
;              entering node from other nodes in comp to zero;
;              Do the same with exiting streams;
;            Endfor;
;            Use one of the tearnode–selection–criteria (which of them is
;            decided by the value of the formal parameter scoring–function)
;            to select the node with the highest score as tearnode in the
;            strong component comp;
;            (the result of this function is a 2–element list where the 1.
;             element is one of the symbols 'in or 'out, indicating whether
;             entering or exiting streams to the tearnode should be torn,
;             and the 2nd element is the number of the selected tearnode)
; End description


(defun sum–up–weights (comp)
  (dolist (node comp nil)
    (dolist (successor (adjacent node) nil)
      (if (member successor comp)
          (outs0 node (+ (outs node) (weight node successor)))))
    (dolist (predecessor (predecessors node) nil)
      (if (member predecessor comp)
          (ins0 node (+ (ins node) (weight predecessor node)))))))
```

```
; Description: For every node node in the strong component comp do
;                For every successor adjacent to node do
;                  If successor is contained in comp then
;                    Add the weight of the stream from node to successor
;                      to the the node'th element of the vector outs;
;                      (which denotes the weighted sum of streams exiting
;                       node and entering other nodes contained in comp)
;                  Endif;
;                Endfor;
;                Do the same with all nodes predecessing node;( --> ins)
;              Endfor;
; End description


(defun liens-scoring-function (node)
  (cond ((< (ins node) (outs node))
         (list 'in (/ (loops node) (ins node))))
        (t (list 'out (/ (loops node) (outs node))))))

;          The new tearnode-selection-criterion :
;          If the weighted number of entering streams to a node is
;              less than the weighted number of exiting streams then
;            the score of the node is equal to the number of unit loops
;              it is contained in divided by the weighted number of
;              entering streams, AND its entering streams might be torn
;          Else the score of the node is equal to the number of loops it's
;              contained in divided by the weighted number of exiting
;              streams, AND its exiting streams might be torn


(defun gundersens-scoring-function (node)
  (list 'in (/ (outs node) (ins node))))

;          The old tearnode-selection-criterion:
;          The score of a node is the weighted number of exiting streams
;          divided by the weighted number of entering streams, AND
;          it's entering streams might be torn
```

```
(defun perform–tearing (type node comp)
  (prog (tear–subset)
    (cond((equal type 'out)
          (dolist (successor (adjacent node)nil)
                (cond ((member successor comp)
                      (setq tear–set (append tear–set (list (list
                                    node successor))))
                      (adjacent0 node (delete successor
                                    (adjacent node)))
                      (predecessors0 successor
                          (delete node (predecessors successor)))))))
          (t
           (dolist (predecessor (predecessors node) nil)
                (cond ((member predecessor comp)
                      (setq tear–set (append tear–set (list (list
                                    predecessor node))))
                      (adjacent0 predecessor (delete node
                                    (adjacent predecessor)))
                      (predecessors0 node
                          (delete predecessor
                              (predecessors node)))))))))
  (number0 node 'blocked))

; Description: If (the tearnode–selection–criterion decided that) the
;                  exiting streams of the tearnode should be torn then
;              remove all streams exiting the tearnode and entering
;                  other nodes contained in the strong component comp
;                  (i.e. delete these nodes from both the adjacency–
;                   and precedence–representations of the graph)
;            Else do the equivalent on the entering streams of the tearnode
;            Endif;
;            Block the tearnode;
; End description


(defun update–unitcycle–information (tearnodes)
  (dolist (tearnode tearnodes nil)
    (dolist (unit–cycle cycle–list nil)
          (cond ((member tearnode unit–cycle)
                (delete unit–cycle cycle–list)
                (dolist (node unit–cycle nil)
                      (loops0 node (– (loops node) 1)))))))
```

```
; Description: For every tearnode in the selected (sub–)set of tearnodes do
;              For every unit–cycle in the list of remaining unit–cycles do
;                If tearnode is contained in unit–cycle then
;                   Remove unit–cycle from the list;
;                   For every node contained in unit–cycle do
;                      Decrement node's associated number of occurrences
;                        in unit–cycles (an element in the vector loops)
;                        with one;
;                   Endfor;
;                Endif;
;             Endfor;
;          Endfor;
; End description


;=============================================================================

; FUNCTIONS FOR IDENTIFICATION OF ONE OR A USER–SPECIFIED NUMBER OF
; PRECEDENCE–ORDERINGS :


; Note !
; In this last part of the algorithm the vectors NUMBER and OUTS have a new
; interpretation (because I wanted to use already allocated space and not
; have to make new vectors) :
;      number(x) is used to store info on whether node x is in a current
;                partial precedence–ordering (marked 'computed)
;                or not (initially marked 'blocked ,from the tearing part
;                of the algorithm. Later marked 'released, when
;                partial orderings are completed)
;      outs(x)   is used to keep a record of how many of x's predecessors
;                in the graph are NOT preceding node x in the current
;                precedence–ordering.(node x will not be available as an
;                extension to a current partial precedence–ordering unless
;                outs(x) = 0)
```

```
(defun precedences (strongcomps precedence–number)
  (enable–restrictions strongcomps)
  (do ((j 0 (+ 1 j)))
      ((> j number–of–nodes))
      (outs0 j (length (predecessors j))))
  (let (w result global–counter)
       (do ((j number–of–nodes (– j 1)))
           ((< j 0))
           (cond ( (available–node 'dummy–parent j)
                   (push j w))))
       (setq global–counter (length w))
       (cond ( (> global–counter precedence–number)
               (setq w (butlast w (– global–counter precedence–number)))))
       (dolist (x w result)
               (setq result (append result
                                    (mapcar '(lambda (y)
                                                     (cons x y))
                                            (backtr x)))))))

; Description:  Build auxiliary datastructures
;               (see description of function enable–restrictions);
;               For j from 1 to number–of–nodes do
;                 outs(j) := the number of predecessors of j in the graph;
;               Endfor;
;               Find all alternative available start–nodes using function
;               available–node;
;               Global–counter := the number of alternative start–nodes
;                  (this variable may be interpreted as the number of
;                   currently foreseen precedence–orderings);
;               If global–counter > number–of–desired–precedence–orderings
;               then skip the superfluous alternatives;
;               For every alternative start–node do
;                 Extend this partial precedence–ordering (of length 1)
;                  using function backtr;
;               Endfor;
;               Result a list of the completed precedence–orderings;
; End description

(defun backtr (node)
  (let (w z result)
       (mark–computed node)
       (setq w (available–list node))
       (cond ( (null w)
               (setq result '(()) ))
             ( t
               (dolist (x w)
                 (setq result (append result
                                      (mapcar '(lambda(y)
                                                      (cons x y))
                                              (backtr x)))))))
       (mark–released node)
       result))
```

```
; Description: Mark node 'computed
;              (it is in a current partial precedence–ordering);
;              Find alternative extensions to the current partial ordering
;                using function available–list;
;              If no possible extentions then
;                Dummy–variable2 := '(());
;              Else
;                Dummy–variable2 := NIL;
;                For every alternative extension do
;                   Dummy–variable1 := (cons extention backtr(extention))
;                   (recursion, in other words)
;                   Dummy–variable2 := (append dummy–variable2
;                                              dummy–variable1)
;                Endfor;
;              Endif;
;              Mark node released; (current ordering is completed)
;              Result Dummy–variable2;
; End description
; (There are things that can be explained in LISP only)


(defun mark–computed (node)
  (number0 node 'computed)
  (dolist (w (adjacent node))
    (cond (w (outs0 w (– (outs w) 1))))))

;       The node is marked 'computed
;       and all its adjacent nodes have one unavailable predecessor less


(defun mark–released (node)
  (number0 node 'released)
  (dolist (w (adjacent node))
    (cond (w (outs0 w (+ (outs w) 1))))))

;       The node is marked released,
;       and all its adjacent nodes have one unavailable predecessor more
```

```
(defun enable-restrictions (strongcomps)
  (do (( j 0 (+ 1 j)))
      (( > j number-of-nodes))
      (setf (get 'strong j) nil))
  (let ((i 0) ins-to-comp)
      (dolist (comp strongcomps)
        (setq i (+ 1 i))
        (setf (get 'comp-index i) comp)
        (setq ins-to-comp nil)
        (dolist (node comp)
          (setf (get 'strong node) i)
          (dolist (pred (predecessors node))
            (cond ((and pred
                        (not (member pred comp))
                        (not (member pred ins-to-comp)))
                   (push pred ins-to-comp)))))
        (setf (get 'comp-ins-index i) ins-to-comp))))


;       It is desirable to prevent premature entrance into or premature exit
;       out of strong components as it may be shown that not preventing this
;       will include nodes in iterative calculations that could have been left
;       outside.  Premature entrance into a strong component is going into it
;       before all nodes in the component have all of their predecessors that
;       are NOT CONTAINED IN THE COMPONENT in the current partial precedence-
;       ordering already.  Premature exit is leaving the component before all
;       nodes in it  are in the precedence-ordering.

;       The above is the motivation for the construction and use of the following
;       datastructures :

;       Comp-index : a property-list under the symbol 'comp-index.
;       The property-names are numbers from 1 to the number of strong components.
;       Corresponding values are the lists of nodes contained in the components.

;       Strong : a property-list under the symbol 'strong with names numbers from
;       0 to number-of-nodes and values showing which one of the numbered strong
;       components this node is contained in.

;       Comp-ins-index : a property-list under the symbol 'comp-ins-index with
;       names numbers from 1 to the number of strong components and values the
;       list of nodes outside the strong component but having streams leading
;       to any node IN the component.
```

```
; Description: Set all properties under 'strong to NIL;
;              i := 0;
;              For every comp in strong components do
;                i := i + 1;
;                Set property named i under 'comp–index equal to comp;
;                Dummy := NIL;
;                For every node in comp do
;                   Set property named node under 'strong equal to i;
;                   For every predecessor of node do
;                      If predecessor is outside comp AND
;                           not already present in Dummy then
;                        include predecessor in Dummy
;                      Endif;
;                   Endfor;
;                Endfor;
;                Set property named i under'comp–ins–index equal to Dummy;
;              Endfor;
; End description


(defun available–list (node)
  (let (result adj temp)
    (setq adj (reverse (adjacent node)))
    (dolist (w adj result)
      (cond ( (and w (available–node node w))
              (push w result))))
    (cond (result)
          (t (do ((j 0 (+ j 1)))
                 ((> j number–of–nodes) result)
                 (cond ( (available–node node j)
                         (push j result))))))
    (cond ( (null result) nil)
          ( (= precedence–number 1)
            (setq result (list (car result))))
          (t (setq temp (+ global–counter (– (length result) 1)))
             (cond ( (<= temp precedence–number)
                     (setq global–counter temp))
                   (t (setq temp (– temp precedence–number))
                      (setq precedence–number 1)
                      (setq result (butlast result temp))))))
    result))

;       First available nodes are sought among the nodes adjacent to
;       the current node.
;       If none is found here ,available non–adjacent nodes to the
;       current node are sought.
;       If the number of foreseen distinct precedence–orderings exeeds
;       the number of desired precedence–orderings, the superfluous
;       alternatives are ignored, and in forthcoming searches only
;       one available node will be returned.
```

```
(defun available–node (parent node)
  (cond ( (not (may–be–available node)) nil)
        ( (eq  (strong parent) (strong node)) t)
        ( (and (single parent) (single node)) t)
        ( (and (single parent) (strong node)) (all–ready (strong node)))
        ( (and (strong parent) (single node)) (all–computed (strong parent)))
        (t
          (and (all–computed (strong parent)) (all–ready (strong node))))))

;       A node is available if it is "may–be–available"
;       (is NOT already in the current partial ordering AND
;        has all its predecessors in the graph already in the ordering)
;       AND parent and node are in the same strong component
;          OR both parent and node are single nodes
;          OR parent is single AND node is in a strong component comp
;             AND all nodes ouside comp having streams leading to any
;                 node in comp are already in the partial ordering
;          OR parent is in a strong component comp AND node is single
;             AND all nodes in comp are already in the partial ordering
;          OR parent is in a strong component comp1 AND
;             node is in a strong component comp2 AND
;             all nodes in comp1 are in the partial ordering AND
;             all nodes outside comp2 having streams leading to any
;             node in comp2 are in the partial ordering


(defun may–be–available (node)
  (and (neq (number node) 'computed)
       (= (outs node) 0)))

;       A node is "may–be–available" if it is NOT already in the
;       partial ordering AND has all its predecessors in the graph
;       already in the ordering


(defun all–computed (comp–index)
  (let ((result t))
    (dolist (x (get 'comp–index comp–index) result)
          (cond ((neq (number x) 'computed) (setq result nil))))))

;       Checks whether all nodes in a component are in the
;       partial ordering.


(defun all–ready (comp–ins–index)
  (let ((result t))
    (dolist (x (get 'comp–ins–index comp–ins–index) result)
          (cond ((neq (number x) 'computed) (setq result nil))))))

;       Checks whether all nodes outside a component having streams to
;       any node in the component are already in the partial ordering

(defun strong (node) (get 'strong node))

;       Returns the number of the strong component the node is contained in
;       or nil if the node is a single node.


(defun single (node) (null (strong node)))
```