A storage scheme for hierarchic structures

W. F. Smyth* and E. Rădăceanu†

The representation of a tree by a right-threaded binary tree, as described for example by Knuth (1968, pp. 332 ff.), is extended to permit representation of 'hierarchic structures' (directed graphs without circuits). This representation corresponds to a compact storage scheme useful both for ascent and descent of the hierarchy.

(Received May 1972)

Hierarchies
We consider here finite connected directed graphs without multiple edges. More precisely, let \( X \) be a finite set and \( \gamma : X \to X \) a mapping which together satisfy the following (connectedness) property:

There exists a sequence \((x_1, x_2, \ldots, x_K)\) containing every element \( x_i \in X \) at least once, such that either \( x_i \in \gamma x_{i+1} \) or \( x_{i+1} \in \gamma x_i \) or both, for each \( i = 1, 2, \ldots, K - 1 \).

Then we define a graph \( G \) to be the pair \((X, \gamma)\). Elements \( x \in X \) are called vertices of \( G \), and pairs \((x, y)\) such that \( y \in \gamma x \) are called edges of \( G \). For any edge \((x, y)\), \( y \in \gamma x \) is called an immediate consequent of \( x \), and \( x \in \gamma^{-1} y \) is called an immediate antecedent of \( y \). More generally, \( y \) is a consequent of \( x \) if \( y \in \Gamma x \sqcup \gamma x \), and \( x \) is an antecedent of \( y \) if \( y \in \Gamma^{-1} y \sqcup \gamma^{-1} y \).

We may now define a hierarchic structure (or hierarchy) \( H \) to be a graph which contains no circuits; that is, such that for each \( x \in X \), there exists no consequent of \( x \) which is also an antecedent of \( x \). A hierarchy then is a finite connected version of Berge's 'graph without circuits' (1958, p. 12). It may also be thought of as the directed equivalent of a forest (of disjoint trees): analogous to the requirement that a tree has no cycles, we require here that \( H \) has no circuits. Or we may think of \( H \) as a generalised arborescence (Berge, 1958, p. 157), which permits any element \( x \) either to have no immediate antecedent (possible multiple root vertices) or to have many (possible cycles).

![Fig. 1 Typical hierarchy](https://example.com/fig1.png)

Examples of hierarchies are: a PERT network; an organisation of agricultural products into additive sets; a structured listing of countries, regions within countries, and geographical groupings or organisations to which the countries or regions belong; an inventory explosion file; a pattern for the grammatical breakdown of English sentences; a computer program (Aho and Ullman, 1972); the syntax of a programming language (Dennis and Patil, 1970; Rosenberg, 1971).

In practice, it is characteristic of many hierarchies that we wish, for a given element \( x \), to be able to locate either its immediate consequents (\( \gamma x \)) or its immediate antecedents (\( \gamma^{-1} x \)). Thus, in practice, we need to store two lists for each element in the graph: one (the consequent list) corresponding to the direction induced by the mapping, the other (antecedent list) corresponding to the reverse direction (inverse mapping).

Fig. 1 shows a typical hierarchy, and Fig. 2 shows the lists which correspond to it. Note that in terms of computer storage, these lists give rise to two alternatives: either we define fixed length lists for each vertex, which usually involves storage of many zero values; or we allow the lists to be variable in length, which requires considerable additional storage for pointers and complicates processing.

Our essential purpose in this article is to present a storage scheme for hierarchies which is, as a rule, considerably more compact than that of Fig. 2 and which at the same time retains the important facility of allowing us to locate both consequents and antecedents quickly (rapid ascent and descent).

Fig. 2 Computer representation of hierarchy

<table>
<thead>
<tr>
<th>Vertex</th>
<th>( \gamma )-List</th>
<th>( \gamma^{-1} )-List</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>02, 10, 11</td>
<td>01</td>
</tr>
<tr>
<td>02</td>
<td>03, 05</td>
<td>02</td>
</tr>
<tr>
<td>03</td>
<td>04</td>
<td>03, 07</td>
</tr>
<tr>
<td>04</td>
<td>05</td>
<td>02, 07</td>
</tr>
<tr>
<td>05</td>
<td>06, 08, 09</td>
<td>07, 10, 11</td>
</tr>
<tr>
<td>06</td>
<td>07</td>
<td>04, 05, 06</td>
</tr>
<tr>
<td>07</td>
<td>08</td>
<td>01</td>
</tr>
<tr>
<td>08</td>
<td>09</td>
<td>10</td>
</tr>
<tr>
<td>09</td>
<td>10</td>
<td>01</td>
</tr>
<tr>
<td>10</td>
<td>06, 08, 09</td>
<td>01, 11</td>
</tr>
<tr>
<td>11</td>
<td>06, 10</td>
<td>01</td>
</tr>
</tbody>
</table>

Notation
The number of vertices of a hierarchy \( H \) will be denoted \( N \), of which \( r \) are root vertices (no antecedents), \( n \) are vertices with multiple antecedents, and \( s \) are vertices with no consequents (leaf vertices). The number of edges of \( H \) will be denoted by \( M \); each vertex \( x \in H \) will have \( m_x \) antecedents (\( m_x = |\gamma^{-1} x| \)).

*International Labour Office, Geneva, Switzerland.
†Academia Ștefan Gheorghiu, Institutul Central de Pregătire și Perfectionare a Cadrelor de Conducere din Economie și Administrația de Stat, Bucharest, Romania.
that $M = \sum_{i=1}^{n} m_i$: the total number of edges ending at the $n$ vertices with multiple antecedents is $m(\geq n)$, so that $m = \sum_{m_i \geq 1} m_i$. Then $M = (N - r) + (m - n)$. Moreover, from the general result for a graph $G$ that $N - 1 \leq M \leq N^2$, we deduce, by considering the necessary conditions for the elimination of circuits, that for a hierarchy $H$,

$$N - 1 \leq M \leq N(N - 1)/2.$$ 

That $M$ may actually attain its upper limit is shown by the example in Fig. 3.

In the example of Fig. 1, $N = 11$, $r = 2$, $n = 4$, $s = 5$; $M = 14$, $m = 9$. The computer representation exemplified by Fig. 2 requires storage of $2M$ (in our case, 28) vertices.

### Storage scheme

We shall describe in this section a storage scheme which avoids variable lists and reduces storage required per edge to two vertex numbers plus three bits for indicators, while at the same time maintaining high processing speed for both ascent and descent of the hierarchy.

We begin by transforming the hierarchy $H$ into a forest $F$. We do this by considering the $n$ vertices with multiple antecedents; corresponding to each such vertex $x_i$, we create $m_i - 1$ new vertices, yielding a set of $m_i$ identified vertices $x_{i}$, $i = 1, 2, \ldots, m_i$; then we assign to each $x_{i}$ one and only one distinct antecedent from the original $m_i$ antecedents of $x$. The new Identified Vertices (IVs) are identified with each other by means of a circular list. See Figs. 4 and 5.

It is not difficult to verify the following results related to the transformation $H \rightarrow F$:

1. The transformation has a well-defined inverse which takes $F \rightarrow H$.

2. The number of disjoint trees in $F$ (also the number of blanks in the column $\gamma^{-1}$ Vertex) is $r$.

3. The number of vertices in $F$ is $N + m - n = M + r$.

We now transform each tree in $F$ into a right-threaded binary tree, using a process described by Knuth (1968, pp. 332 ff). The result is a binary forest $B$ such as illustrated in Fig. 6.

In Fig. 6 we have omitted for the sake of simplicity indication of the identities among the vertices; these are however shown in the IV column of Fig. 7, which illustrates the computer representation of $B$. Note that in Fig. 7 we show the directions opposite to those of Fig. 6: we are making use of the fact that each vertex has at most one antecedent in order to save computer storage. Note also in Fig. 7 that the number of entries in the Thread column ($N - s$) is equal to the number of groups of entries in the $\gamma$-List column of Fig. 5.

What we have done here in effect is to replace all but (the lefthand) one of each set of antecedent-consequent relationships with relationships between immediate consequents of the

<table>
<thead>
<tr>
<th>Vertex</th>
<th>$\gamma$-List</th>
<th>$\gamma^{-1}$ Vertex</th>
<th>IV</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>02, 10, 11</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>02</td>
<td>03, 05</td>
<td>02</td>
<td>02</td>
</tr>
<tr>
<td>03</td>
<td>04</td>
<td>03</td>
<td>12</td>
</tr>
<tr>
<td>04</td>
<td>02</td>
<td>02</td>
<td>13</td>
</tr>
<tr>
<td>05</td>
<td>06</td>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>06</td>
<td>04, 05, 06</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>07</td>
<td>10</td>
<td>09</td>
<td>10</td>
</tr>
<tr>
<td>08</td>
<td>09</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>09</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>06, 08, 09</td>
<td>01</td>
<td>16</td>
</tr>
<tr>
<td>11</td>
<td>06, 10</td>
<td>01</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>07</td>
<td>07</td>
<td>04</td>
</tr>
<tr>
<td>13</td>
<td>07</td>
<td>07</td>
<td>05</td>
</tr>
<tr>
<td>14</td>
<td>07</td>
<td>07</td>
<td>15</td>
</tr>
<tr>
<td>15</td>
<td>11</td>
<td>11</td>
<td>06</td>
</tr>
<tr>
<td>16</td>
<td>11</td>
<td>11</td>
<td>10</td>
</tr>
</tbody>
</table>

**Fig. 4** Forest corresponding to hierarchy

**Fig. 5** Computer representation of forest

**Fig. 6** Binary trees corresponding to forest

*That is, each $x_i$ represents the same vertex $x$, but now each $x_i$ has only one antecedent.
same antecedent; the addition of the Thread connection then enables us to represent the hierarchy in terms of the series of tangential loops illustrated in Fig. 6. Appendix 1 contains a more formal specification of an algorithm \( FB \) which transforms \( F \rightarrow B \) (ignoring identifications among vertices). With respect to this transformation we may now state the following results:

4. The transformation has a well defined inverse which takes \( B \rightarrow F \).

5. There are \( (M + r) - (N - s) = m - n + s \) blanks in the Thread column; there is a blank at each of \( s \) leaf nodes and at each of the \( m - n \) new vertices created corresponding to the multiple antecedents of \( H \).

6. There are \( m \) entries in the IV column.

Results 5 and 6 together give us the important property:

7. The number of blanks in the Thread column is greater than or equal to the number of entries in the IV column if and only if \( s \geq n \).

Our storage scheme consists then of making use of the property \( s \geq n \), whenever it holds, to combine the Thread and IV columns into one (which we refer to as the TIV column, composed of TIV vertices). This merging of the two columns corresponds to a transformation \( B \rightarrow B' \), which will be described below.

First, however, a word about the essential condition \( s \geq n \). Although it will of course not always hold (see for example Fig. 3), nevertheless we may expect that in many cases of practical interest it will be satisfied. For example, the condition is satisfied if the only vertices with multiple antecedents are also leaf vertices. More generally, we may expect that hierarchies with relatively many leaf vertices and not ‘too’ complicated a structure would be suitable for application of the storage scheme.

Quantitatively, if we denote by \( m_2 \) the maximum length of the \( \gamma^{-1} \)-List corresponding to any vertex, then

\[
M = (N - r) + (m - n)
\]

\[
\leq (N - r) + (m_2 - 1)n
\]

\[
\leq (N - r) + (m_2 - 1)s
\]

provided \( s \geq n \). Hence the value

\[
M_{\text{max}} = (N - r) + (m_2 - 1)s
\]

represents the largest value of \( M \) consistent with the condition \( n \leq s \). Conversely, the smallest value \( M_{\text{min}} \) of \( M \) consistent with the condition \( n > s \) may be calculated by adding \( s + 1 \) additional edges to the original \( N - r \) edges required to satisfy the connectivity assumption; thus,

\[
M_{\text{min}} = (N - r) + (s + 1).
\]

We see that there are three cases: for \( M > M_{\text{max}} \), the storage scheme can definitely not be used; for \( M < M_{\text{min}} \), the storage scheme can always be used regardless of the way in which the vertices are connected; and for \( M_{\text{min}} \leq M \leq M_{\text{max}} \), the storage scheme may or may not be feasible depending on the nature of these connections.

Note that for small \( s \), any necessity to assume that \( M \) is less than or even close to \( M_{\text{min}} \) will probably be rather restrictive. On the other hand, for the not uncommon case \( s \approx N/2 \), we have

\[
M_{\text{min}} \approx 3N/2 - r + 1,
\]

and the condition \( M < M_{\text{min}} \) would be satisfied by a substantial number of the hierarchies encountered in practice.

Note also that for any hierarchy subject to unpredictable change, it would be unwise to make use of the storage scheme if \( M \) were at all close to \( M_{\text{max}} \). Nevertheless, taking \( m_2 = 3 \) and assuming as above that \( s \approx N/2 \), we find

\[
M_{\text{max}} \approx 2N - r,
\]

so that at the lower end of the interval

\[
3N/2 + 1 \leq M + r \leq 2N,
\]

we might expect to be able to make use of the storage scheme safely enough.

In the example of Fig. 1,

\[
M_{\text{min}} = 15, M_{\text{max}} = 19, M = 14;
\]

and in fact \( M \) could be increased by at least 1 (\( N \) of course constant) without precluding use of the storage scheme.

Returning now to the transformation \( B \rightarrow B' \) we may describe it in the following way:

Suppose vertex \( i \) has a Thread vertex, but no IV vertex. Then the TIV vertex for \( i \) is simply the Thread vertex. If \( i \) has an IV vertex, but no Thread vertex, then the TIV vertex is the IV vertex marked by an asterisk (*). If \( i \) has both a Thread vertex and an IV vertex, the TIV vertex is set equal to the Thread vertex and marked by a vertical arrow (↑); the IV vertex of \( i \) is then placed in the first blank prior TIV position and marked by another special sign (+).

The above transformation will always operate successfully provided \( s \geq n \), and provided also we agree to process the TIV column in increasing order of vertex \( i = 1, 2, \ldots \), and to regard it as a circular list (the \( (M + r) \)th TIV vertex immediately precedes the first one). Note, moreover, that the following rule will always enable us to locate the \( \pm \)-vertex corresponding to a given \( \uparrow \)-marker:

Set a counter \( c \) to +1. Search backwards through the TIV column beginning at the position of the \( \pm \)-vertex. For each \( \uparrow \)-marker encountered increment \( c \) by one; for each \( \pm \)-vertex encountered decrement \( c \) by one. The first \( \pm \)-vertex which causes \( c \) to go to zero is the IV vertex corresponding to the original \( \uparrow \)-marker.

Use of this rule permits us to assert that:

8. the transformation has a well defined inverse which takes \( B' \rightarrow B \).

Fig. 8 shows the compacted storage scheme \( B' \) which corresponds to the example of Figs. 6 and 7. Appendix 2 contains a specification of the algorithm \( HB' \) which combines the transformations \( H \rightarrow F, F \rightarrow B, B \rightarrow B' \) into a single transformation \( H \rightarrow B' \). It follows from results (1), (4) and (8) that there exists a well defined inverse transformation \( B' \rightarrow H \). Appendix 3 exhibits the algorithm Findante (i), which makes use of the storage scheme \( B' \) to locate, in an efficient manner, all the antecedents of a given vertex \( i \in H \).

Storage scheme variants

Among the possible modified storage schemes which suggest themselves are the following:

The Computer Journal
<table>
<thead>
<tr>
<th>Vertex</th>
<th>Antecedent</th>
<th>TIV</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>= 01</td>
<td>10</td>
</tr>
<tr>
<td>02</td>
<td>= 02</td>
<td>05</td>
</tr>
<tr>
<td>03</td>
<td>= 03</td>
<td>04</td>
</tr>
<tr>
<td>04</td>
<td>= 03</td>
<td>*</td>
</tr>
<tr>
<td>05</td>
<td>+ 03</td>
<td>*</td>
</tr>
<tr>
<td>06</td>
<td>= 10</td>
<td>*</td>
</tr>
<tr>
<td>07</td>
<td></td>
<td></td>
</tr>
<tr>
<td>08</td>
<td>+ 06</td>
<td></td>
</tr>
<tr>
<td>09</td>
<td>+ 08</td>
<td>≡</td>
</tr>
<tr>
<td>10</td>
<td>+ 11</td>
<td>↑</td>
</tr>
<tr>
<td>11</td>
<td>+ 02</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>+ 13</td>
<td>*</td>
</tr>
<tr>
<td>13</td>
<td>+ 14</td>
<td>*</td>
</tr>
<tr>
<td>14</td>
<td>= 07</td>
<td>*</td>
</tr>
<tr>
<td>15</td>
<td>= 11</td>
<td>*</td>
</tr>
<tr>
<td>16</td>
<td>+ 15</td>
<td>*</td>
</tr>
</tbody>
</table>

Fig. 8 Storage scheme for hierarchies

Variant A
Make use of blanks in the Antecedent column as well, in particular for storage of + vertices (no root vertex may have a + vertex). This provides greater flexibility (the variant is feasible under the relaxed condition \( r + s \geq n \)) at the cost of possible increased search time (both Antecedent and TIV columns must now be searched). Variant A will be attractive in cases where there are many root nodes and also many multiple antecedents; for example, a hierarchy of countries, regions within countries, and organisations of countries and regions.

Variant B
Renumber the vertices of \( H \) so that after each vertex \( x \) with \( m_x - 1 \) antecedents, the subsequent \( m_x - 1 \) vertices are exactly those vertices (the IVs) identified with \( x \). This variant reduces the storage required by only one bit per edge, but eliminates all searches. Moreover, the system may be used even when \( n > s \). Unfortunately, however, it is usually not possible to use variant B, because to communicate with its environment, the computer system must usually make use of the originally defined vertex numbers; this will normally require storage of a third column of vertex numbers, which are used to convert external numbers to computer numbers, and vice versa. A related disadvantage is that any change to \( H \) will in general require renumbering of all vertices and in addition a corresponding restructuring of storage.

Storage and timing considerations
1. It is not difficult to calculate that the total number \( T \) of vertex numbers (excluding blanks) which are required by the above storage scheme is

\[
T = 2M + r - (s - n)
\]

a value which is not necessarily equal to the \( 2M \) vertex numbers required by the original storage scheme (Fig. 2). It follows therefore that the transformation \( H \rightarrow B' \) cannot be represented by simple rearrangement of the rows and columns of the incidence matrix corresponding to \( H \). Note that when \( s > r + n \), we even have \( T < 2M \); in other words, it is possible not only to organise storage more economically using the transformation \( H \rightarrow B' \), but also to reduce the number of non-blank vertex numbers which must be stored.

2. Let \( d \) represent the number of decimal digits in \( M + r \); that is, in the highest vertex number. Then the scheme described above requires storage of

\[
D = (M + r)(2d + 1)
\]
decimal digits, where we have allowed one decimal digit (actually three bits suffice) for indicators (+, =, *, etc.). In the case \( d = 3 \), we require about seven decimal digits storage for each edge; in packed form, this is less than one IBM 360 word.

By contrast, the original storage scheme required (in fixed list length format)

\[
D' = Nd' (m_1 + m_2)
\]
decimal digits, where now \( d' \) is the number of decimal digits in \( N(d - 1 \leq d' \leq d) \), and \( m_1 \) and \( m_2 \) are the maximum length of the \( y \) and \( y^{-1} \) lists, respectively. Note that \( m_1 + m_2 > 2 \) in all but trivial cases. Note moreover that in most cases of practical interest, \( D < D' \). That this is not always true, however, is shown by Fig. 9.

3. On the whole it may be anticipated that for most applications which require both rapid ascent and descent, the proposed storage scheme will yield processing speeds for hierarchies which are about the same as those associated with the original storage scheme. It will be necessary to search the TIV column only when the given vertex is marked ↑; that is, only once corresponding to each non-leaf vertex with multiple antecedents. If all vertices with multiple antecedents are also leaf vertices, no searches will be necessary.

4. Note that the algorithm \( HB' \) presented in Appendix 2 requires two passes of the hierarchy in ascending sequence of vertex number \( i \) (traversing in all \( 2N \) vertices) to transform \( H \) into \( B' \). The first pass (the antecedent phase) may however normally be carried out in the time required to input \( H \); thus, the extra cost of using the storage scheme is represented by the consequent phase. The consequent phase may in fact be shortened somewhat (the search in step 2.5 could be eliminated), but apparently only at the cost of making the antecedent phase incompatible with simultaneous input of \( H \).

The algorithm \( HB' \) has two principal disadvantages: in the consequent phase it requires the consequents of every vertex of \( H \) to be simultaneously addressable (step 2.3); and, as a result of the ↑ indicator system, it depends upon sequential processing of the vertices.

Appendix 1 Algorithm FB

1 [Initialise.] \( i \leftarrow 0 \).
2 [All vertices considered?] \( i \leftarrow i + 1 \). If \( i > M + r \), goto 7.
3 [No antecedents?] If \( \gamma^{-1}i = \phi \), ANTECEDENT(\( i \)) ← (blank, 0).
4 [No consequents?] Let \( \gamma i = (x_1, x_2, \ldots, x_k) \). If \( K = 0 \), THREAD(\( i \) ← 0, goto 2; otherwise, ANTECEDENT(\( x_k \)) ← (\( , i \)), \( k \leftarrow 1 \).
5 [All consequents?] \( k \leftarrow k + 1 \). If \( k > K \), THREAD(\( i \) ← \( x_k \), goto 2.
6 [Antecedent of \( x_k \)] ANTECEDENT(\( x_k \)) ← (\( , \), \( x_{k-1} \)).
Goto 5.
7 [Exit.] Algorithm terminates.

Appendix 2 Algorithm HB'

Before beginning the algorithm itself, we define three useful functions:

\[
\begin{align*}
\text{left}(a, b) & \equiv a; \text{that is, the lefthand member of the ordered pair } (a, b); \\
\text{right}(a, b) & \equiv b;
\end{align*}
\]

\[
\begin{align*}
M + r & = 10, d = 2 \\
D & = 50 \text{ digits}
\end{align*}
\]

\[
\begin{align*}
N & = 7, d' = 1, m_1 = 3, \\
m_2 & = 2 \quad D' = 35 \text{ digits}
\end{align*}
\]

Fig. 9 Original storage scheme superior
jvertex \( (H, i) \) \( = \) a vertex number in the hierarchy \( H \) dependent on the choice of the initial vertex number \( i \); that is, the first value \( j \) such that \( \text{SUM} = 0 \), when initially \( \text{SUM} \leftarrow 1 \) and \( j \) subsequently takes the values

\( i - 1, i - 2, \ldots, 1, N, N - 1, \ldots, i + 1, \)

when for each value of \( j \)

\( \text{SUM} \leftarrow \text{SUM} + 1 \) if \( |y| > 0 \) and \( |y - 1| > 1 \),

\( \text{SUM} \leftarrow \text{SUM} - 1 \) if \( |y| = 0 \) and \( |y - 1| \leq 1 \).

1.1 [Initialise for antecedent phase.] \( i \leftarrow 0, N' \leftarrow N \).

1.2 [All vertices considered?] \( i \leftarrow i + 1 \). If \( i > N \), goto 2.1.

1.3 [Clear.] Let \( y^{-1} = (x_1, x_2, \ldots, x_k) \). If \( K = 0 \),

\( \text{ANTECEDENT}(i) \leftarrow (\text{blank}, 0) \); otherwise,

\( \text{ANTECEDENT}(i) \leftarrow (x_1) \). If \( K \leq 1 \),

\( \text{TIV}(i) \leftarrow (\text{blank}, 0) \), goto 1.2.

1.4 [Provisional IV for vertex \( i \).] \( i \leftarrow 1 \).

\( \text{TIV}(i) \leftarrow (x_1, x_2 \ldots, x_k) \).

1.5 [TIVs and ANTECEDENTs for copies of \( i \).] \( k \leftarrow k + 1 \),

\( N' \leftarrow N' + 1 \). ANTECEDENT\((N') \leftarrow (x_k) \).

If \( k < K \), \( \text{TIV}(N') \leftarrow (x_1, x_2 \ldots, x_k) \), goto 1.4; otherwise,

\( \text{TIV}(N') \leftarrow (x_1, \ldots, x_k) \), goto 1.2.

2.1 [Initialise for consequent phase.] \( i \leftarrow 0 \).

2.2 [All vertices considered?] \( i \leftarrow i + 1 \). If \( i > N \), goto 2.8.

2.3 [Consequences of \( i \).] Let \( y = (x_1, x_2, \ldots, x_k) \).

If \( K = 0 \), goto 2.2; otherwise, \( k \leftarrow 0 \).

2.4 [All consequences considered?] \( k \leftarrow k + 1 \), \( p \leftarrow x_k \).

If \( k > K \), goto 2.7.

2.5 [Find \( p \) such that \( i = \text{right}(\text{ANTECEDENT}(p)) \).]

If \( \text{right}(\text{ANTECEDENT}(p)) \neq i \), (if left \( \text{TIV}(p) \neq \)*,

\( p \leftarrow \text{vertex}(H, p) \), \( p \leftarrow \text{right}(\text{TIV}(p)) \), goto 2.5.

2.6 [Modify \( \text{ANTECEDENT}(p) \).]

If \( k > 1 \), \( \text{ANTECEDENT}(p) \leftarrow (+, q) \).

\( q \leftarrow p \).

Goto 2.4.

2.7 [Establish THREAD(i).] If \( \text{right}(\text{TIV}(i)) = 0 \),

\( \text{TIV}(i) \leftarrow (\text{blank}, q) \); otherwise,

\( \text{TIV}(\text{vertex}(H, i)) \leftarrow (+, \text{right}(\text{TIV}(i))) \),

\( \text{TIV}(i) \leftarrow (\dagger, q) \). Goto 2.2.

2.8 [Exit.] Algorithm terminates.

**Appendix 3 Algorithm Findante** \( (i) \)

Given a storage scheme \( B' \) (Fig. 8) corresponding to a hierarchy \( H \),

To find all antecedents of a given vertex \( i \in H (i < N) \).

At the conclusion of the algorithm the vector ANTELIST specifies all the antecedents of \( i, h \) in number, with no duplicates.

The algorithm makes use of temporary storage vectors

BITPATTERN and IVLIST: BITPATTERN consists of \( N \) 0/1 indicators used to indicate which vertices have already

been found to be antecedents of \( i \), while IVLIST may contain up to \( m - n \leq (m_2 - 1) \) Identified Vertices associated with \( i \) or its antecedents. For definitions of the functions left \( (a, b) \),

right \( (a, b, c) \), and jvertex \( (H, i) \), see Appendix 2; note however that in terms of the storage scheme \( B \),

the computation of the value SUM used to define jvertex should be re-expressed so that for each value of \( j \),

\( \text{SUM} \leftarrow \text{SUM} + 1 \) if \( \text{left} \( \text{TIV}(j) \) \) \( = \dagger \),

\( \text{SUM} \leftarrow \text{SUM} - 1 \) if \( \text{left} \( \text{TIV}(j) \) \) \( = \nabla \).

1 [Initialise.] \( h \leftarrow 0 \), \( K \leftarrow 0 \), \( k \leftarrow 0 \).

Set every position in BITPATTERN to zero.

2 [Test for root vertex.] If \( \text{ANTECEDENT}(i) = (\text{blank}, 0) \),

\( \text{go to} \ 7 \).

3 [Locate the Identified Vertices of \( i \).] If \( \text{left} \( \text{TIV}(i) \) \) \( = \ast \),

\( i \leftarrow \text{right} \( \text{TIV}(i) \) \); if \( \text{left} \( \text{TIV}(i) \) \) \( = \dagger \),

\( i \leftarrow \text{right} \( \text{TIV}(\text{vertex}(H, i)) \) \); otherwise, goto 5.

4 [Store the Identified Vertices of \( i \).] If \( i > N \), \( K \leftarrow K + 1 \),

\( \text{IVLIST}(K) \leftarrow i \), goto 3.

5 [Identify the immediate antecedent of \( i \).] If \( \text{left} \( \text{ANTECEDENT}(i) \) \) \( = \ast \), \( i \leftarrow \text{right} \( \text{ANTECEDENT}(i) \) \),

\( \text{go to} \ 5 \); otherwise, \( i \leftarrow \text{right} \( \text{ANTECEDENT}(i) \) \).

6 [Store immediate antecedent in ANTELIST and BITPATTERN.] If \( \text{BITPATTERN}(i) = 0 \),

\( \text{BITPATTERN}(i) \leftarrow 1, h \leftarrow h + 1 \), \( \text{ANTELIST}(h) \leftarrow i \),

\( \text{go to} \ 2 \).

7 [Run of antecedents ended; pick up Identified Vertex.] If \( k < K \), \( k \leftarrow k + 1 \), \( i \leftarrow \text{IVLIST}(k) \), goto 5.

8 [Exit.] Algorithm terminates.

Note that the algorithm may be modified to suit various purposes.
It might be desirable, for example, to make use of the fact that the number of times Step 7 is entered is exactly the total number of distinct paths leading from root vertices of \( H \) to \( i \).
Moreover, modification of BITPATTERN and Step 6 would permit a count to be kept in BITPATTERN of the number of distinct paths leading from any vertex \( k \) to vertex \( i \).
Alternatively, elimination of BITPATTERN and modification of Step 6 to read simply

\( h \leftarrow h + 1 \), \( \text{ANTELIST}(h) \leftarrow i \)

converts Findante(i) into an algorithm for finding all immediate antecedents of \( i \).
Note also that the storage of vertices in ANTELIST is in ascending runs of immediate antecedents,

each run ending either at a root vertex or just before it merges into a previously stored run.

The corresponding algorithm Findcon(i), which finds all the consequences of a given vertex \( i \), is similar in structure to

Findante(i), but somewhat simpler, as a result of the fact that every Identified Vertex is also in effect a leaf vertex.

**References**


