Stupid PATHLOC tricks

Joe English

23 Oct 1995

Introduction

The path location address or "pathloc" form is one of HyTime's more obscure features. The HyTime tutorials that I've seen ([2], [3]) explain the pathloc mechanism, but not the motivation behind it: just what is pathloc good for?

This article explores some ways in which an SGML engine can use path location addresses to simplify tree processing.

> *This document contains a few editorial notes and questions, such as this one. If you can answer any of these questions, please let me know!*

Fundamental concepts

Double-check this

In HyTime, a *marker* is a non-zero integer which designates a location or an extent along some axis.

A *dimension specification* or *dimspec* is a pair of markers which locates a range along an axis. If both markers in a dimspec are positive, then the first marker specifies the beginning of the range (position) and the second specifies the length of the range (extent). Indexing is 1-based; for example, the dimspec `1 1` locates the first element in a range and `3 7` locates the third through tenth elements.

> There are other interpretations if one or both markers are negative; see [1] p. 106, [3] pp. 3-4, or section 5 of [2] for details. For the purposes of this article, only the case where both markers are positive is needed.

A *dimlist* is a list of dimspecs, or a list of markers which are paired up to form dimspecs. A dimlist can be used to locate a rectangular region of a multi-dimensional coordinate space, with one dimspec per coordinate axis.

> In other contexts, dimlists locate *multiple* regions along a *one-dimensional* coordinate space.

Question: is "locate" the correct term here?

Finally, a *path location address* is a dimlist which locates part of a tree, by treating the tree as a 2-dimensional matrix. A path location address contains two dimspecs, one for the columns and one for the rows. That is, it's a pair of pairs of markers $((c,n),(r,m))$ where, assuming all markers are positive,

- *c* is the first column in the range,
- *n* is the number of columns,
- *r* is the first row, and
- *m* is the number of rows.

In the matrix view of a tree, there is one column for each leaf node, and each column consists of all the elements on the path from the root to the leaf. If the tree is of non-uniform height, "short" columns are padded at the bottom with empty entries.

The tree representation of an SGML document instance has one node for each element, and data content is represented by *pseudo-element* or *PEL* nodes. PEL nodes and empty elements make up the leaves of the tree.

An example

The following HTML document instance

```
<HTML>
<HEAD>
<TITLE>Example</TITLE>
</HEAD>
<BODY>
<P>Text.
<P>Some <EM>more</EM> text.
<P>End.
</BODY>
</HTML>
```

has this tree representation

```
<HTML>
 |--<HEAD>
 |    \--<TITLE>
 |        \--PEL "Example"
 \--<BODY>
     |--<P> (1)
     |    \--PEL "Text."
     |--<P> (2)
     |    |--PEL "Some "
     |    |--<EM>
     |    |    \--PEL "more"
```

```
|   \--PEL " text."
\--<P> (3)
    \--PEL "End."
```

and its path list matrix is

```
row\col| 1       2       3       4       5       6
--------------------------------------------------------
     1 | HTML    HTML    HTML    HTML    HTML    HTML
     2 | HEAD    BODY    BODY    BODY    BODY    BODY
     3 | TITLE   P(1)    P(2)    P(2)    P(2)    P(3)
     4 | Example Text.   Some    EM      text.   End.
     5 | -       -       -       more    -       -
```

An alternate view of the path list matrix is

```
+-------+-----------------------------------------------+
|row\col|   1   |   2   |   3   |   4   |   5   |   6   |
|-------+-------+---------------------------------------|
|   1   |                     HTML                      |
|-------+-------+---------------------------------------|
|   2   | HEAD  |                 BODY                  |
|-------+-------+-------+-----------------------+-------|
|   3   | TITLE | P(1)  |         P(2)          | P(3)  |
|-------+-------+-------+-------+-------+-------+-------|
|   4   |Example| Text. | Some  |  EM   | text. | End.  |
|-------+-------+-------+-------+-------+-------+-------|
|   5   |   -   |   -   |   -   | more  |   -   |   -   |
+-------+-------+-------+-------+-------+-------+-------+
```

Note that start- and end-tags do not appear in this representation, nor does any other markup. The element nodes in the tree may be addressed by the following path location addresses:

```
<HTML>  -- 1 6 1 5
<HEAD>  -- 1 1 2 3
<TITLE> -- 1 1 3 2
<BODY>  -- 2 5 2 4
<P> (1) -- 2 1 3 2
<P> (2) -- 3 3 3 3
<EM>    -- 4 1 4 2
<P> (3) -- 6 1 3 2
```

So what is pathloc good for?

Pathloc can locate a single node in the tree, a complete subtree, or a contiguous range of nodes:

```
PATHLOC(DOMROOT 4 1 4 1)         -- just the EM element node
PATHLOC(DOMROOT 2 1 3 2)         -- second P element and its content
PATHLOC(DOMROOT 2 5 3 3)         -- all three P elements
```

Any contiguous range may be addressed, including those which overlap element boundaries:

```
PATHLOC(DOMROOT 1 3 1 5)         -- all nodes before the EM element
PATHLOC(DOMROOT 5 2 1 5)         -- all nodes after the EM element.
PATHLOC(DOMROOT 4 3 2 4)         -- "more text. End", BODY, P(2), EM, P(3)
```

Pathloc can locate lists of interior nodes as well:

```
PATHLOC(DOMROOT 3 1 1 2)         -- ancestors of P(2)
PATHLOC(DOMROOT 2 5 3 1)         -- children of BODY: P(1), P(2), P(3)
```

In fact, all of the HyTime relational location forms can be implemented on top of pathloc; see below.

And, of course, pathloc can address paths from the root node to any leaf:

```
PATHLOC(DOMROOT 4 1 1 5)         -- HTML, BODY, P(2), EM, and PEL "more"
```

Question: In the HyQ examples, should the first parameter (query domain) be `DOMROOT` *or* `DOMTREE`*? [3] uses* `DOMTREE`*, but [2] suggests that the query domain is a single node.*

Question: Is the syntax right? [3] encloses the marker list in parentheses, while [2] and [1] do not.

Path location addresses are compact: Unlike tree location addresses, they are always four markers long, regardless of the number of nodes addressed or how deep in the tree the range appears.

Pathloc could be particularly useful for HTML 2.0 and other "flat" document types. HTML currently has no hierarchical section elements, only ranked heading elements H1 through H6. Pathloc could be used to locate ranges such as "everything between the third and fourth H1 elements," which would correspond to the third major "section" of an HTML document. (See below for an algorithm to construct such addresses.)

A different interpretation

Rectangular regions of a path list matrix are a somewhat unnatural way to address nodes in a tree. This section defines an alternate interpretation of path location addresses which is more relevant to the underlying tree structure.

First, observe that a tree node may have several distinct path location addresses. For example, all of the following addresses locate the second P element node in the above example:

```
PATHLOC(DOMROOT 3 1 3 1)
PATHLOC(DOMROOT 3 2 3 1)
PATHLOC(DOMROOT 3 3 3 1)
PATHLOC(DOMROOT 4 1 3 1)
PATHLOC(DOMROOT 4 2 3 1)
PATHLOC(DOMROOT 5 1 3 1)
```

In the following definitions, one range is considered "smaller" than another if (in order of priority) it spans fewer columns, spans fewer rows, starts at an earlier column, or starts at an earlier row. In other words, let path location addresses be compared lexicographically by the second, fourth, first, and third markers (assuming all markers are positive).

> Intuitively, ranges are compared based on the number of cells addressed, and equal-sized ranges are compared by their starting position in the path list matrix.

Define the *subtree position* of a node as the smallest path location address which locates that node and all of its descendants. Define the *node address* of a node as the smallest path location address which locates that node and no others.

For example, the subtree position of the second P element is `3 3 3 3` and its node address is `3 1 3 1`. Note that the node address is always the same as the subtree position with the second and fourth markers set to one.

> *Question: which, if either of these is the HyTime "document position" or* `docpos` *property?*
> `<HyPD psn=HTsem2 HyTime pn=docpos>pathloc document position</HyPD>`

Last, assign a *path number* to each leaf node in the tree, starting at one and incrementing the path number for consecutive leaves.

The subtree position of any node can then be viewed as a 4-tuple (p, w, d, h), with

p **("path")** the path number of the first leaf node in the subtree,

w **("width")** the total number of leaves in the subtree;

d **("depth")** the distance from the root to the top of the subtree;

h **("height")** the maximum distance from the top of the subtree to any leaf node.

The (p, w, h, d) 4-tuple is simply a path location address relative to the document root with all markers positive. The path number and width fields (p, w)make up the first dimspec. (first column and number of columns). The depth and height fields (d, h) make up the second (first row and number of rows).

Tree properties

The subtree positions of tree nodes have several easily-verified properties.

- If x is a child of y, then $x.d = y.d + 1$.
- If x is a descendant of y, then $x.d >= y.d$.
- If x is a descendant of y, then $x.w >= y.w$
- If x is the immediate right (following) sibling of y, then $x.p = y.p + y.w + 1$.

- If x is a descendant of y, then $y.p <= x.p <= x.p + x.w <= y.p + y.w.$ (The converse is also true, though this is not immediately obvious.)
- If (p, w, d, h) is the subtree position of a node x, then the path location address $(p', w', d, 1)$ locates x and no other nodes if and only if $p <= p' <= p' + w' <= p + w$.

Tree representation

In the following algorithms, it is assumed that a "generalized binary tree" representation is used to store the document instance, in which each node contains a pointer to its first child and a pointer to its next sibling:

```
type node;
 type nodePtr is access node;
 type node is
 record
     child :       nodePtr;          -- pointer to first child
     next :        nodePtr;          -- pointer to next sibling
     parent :      nodePtr;          -- back-pointer to parent node
     p, w, d, h : integer;           -- pathloc document position of subtree
     -- ... other properties: generic identifier, attributes, etc.
 end;
```

For convenience, it is also assumed that each node contains a back-pointer to its parent node. This simplifies some of the following algorithms, although most could easily be modified to work with different representations (e.g., a nested list representation in Scheme or Lisp.) The main idea is that each node is augmented with its subtree position as defined above.

An algorithm for constructing such a tree from a stream of ESIS events is given below.

The `pathloc` data structure is used to represent path location addresses:

```
type pathloc is record               -- path location address
     p, w, d, h : integer;           -- path number, width, depth, and height
 end;
```

The "before" relation

DSSSL [4] defines the "before" relation as:

> A node *x* is before a node *y* if *x* is not equal to *y* and either *y* is a member of the subtree of *x*, or there is a node *a* such that there are nodes *b* and *c* such that *b* and *c* are children of *a*, *b* precedes *c* among the children of *a*, *x* is in the subtree of *b*, and *y* is in the subtree of *c*.

(Actually this is a slightly simplified definition, as it does not deal with attribute nodes or multiple document trees.)

An equivalent inductive definition is:

- A node is before all its descendants;

- Earlier siblings are before later siblings;
- If *x* is before *y* and *y* is before *z* then *x* is before *z*.

And another equivalent definition is that *x* is before *y* if and only if *x* is visited before *y* in a preorder traversal of the tree. None of these definitions suggest an efficient algorithm to test whether one node is before another.

One approach would to find the second-least common ancestors of *x* and *y* (a tricky problem in itself! But see below.) and comparing their relative positions among the siblings of the least common ancestor. Another is to perform a preorder traversal, stopping when either *x* or *y* is reached.

However, if the subtree position is stored at each node, this can be accomplished with simple arithmetic comparisons:

```
function before(x, y : nodePtr) return boolean is
    return x.p <= y.p
        or x.p = y.p and x.d < y.d
end before;
```

This is much more useful than it sounds. While it is not very often necessary to test if one node is "before" another in an SGML processing application, both DSSSL and HyTime make extensive use of *node sets* and ordered *node lists*. If a total order can be defined on the domain – such as the "before" relation – then it is much easier to create an efficient implementation of the "set" abstract data type. (For example, a sorted list representation could be used, and the **union**, **intersection**, and **difference** operations implemented with a merge-join algorithm, taking $O(N + M)$ time.) An added bonus is that the "before" relation corresponds to the "natural order" for iterating over node sets.

Tree traversal algorithms

The **traverse** function implements a simple preorder traversal:

```
procedure traverse(n : nodePtr) is
    process(n);
    n := n.child;
    while n /= NULL loop
        process(N)
        n := n.next;
    end
end traverse
```

The **pathtrav0** function processes all nodes which are selected by a given pathloc address. It works just like **traverse**, except that it also checks each node against a path location address and only processes those which are in the specified range.

```
procedure pathtrav0(n : nodePtr, l:pathloc) is
    if intersects(n,l) then
        if inrange(n, l) then
            process(n);
        end if;
        n := n.child;
        while n /= NULL loop
```

```
                pathtrav0(n,l);
                n := n.next;
            end loop
        end if
    end traverse
```

The **intersects** function tests if a node's subtree intersects a pathloc range; **inrange** tests if the node itself is included in the range. Note that the two functions are nearly identical, differing only in the depth and height comparisons.

```
    function intersects(n : nodePtr, l : pathloc) return boolean is
        return  l.p <= n.p + n.w
             and n.p <= l.p + l.w
            and l.d <= n.d + n.h
            and n.d <= l.d + l.h;
    end intersects

    function inrange(n : nodePtr, l : pathloc) return boolean is
        return  l.p <= n.p + n.w
             and n.p <= l.p + l.w
            and l.d <= n.d
            and n.d <= l.d + l.h;
    end inrange;
```

pathtrav0 can be improved to take advantage of certain tree properties and prune the search:

```
    procedure pathtrav(n : nodePtr, l:pathloc) is
        if intersects(n,l) then
            if l.d <= n.d then              -- node in range
                process(n);
            end if;
            if n.d + 1 <= l.d + l.h then              -- children may be in range
                n := n.child;
                -- find first child in range:
                while n /= NULL and n.p + n.w < l.p loop
                    n := n.next;
                end loop;
                -- traverse all children in range:
                while n /= NULL and l.p + l.h <= n.p loop
                    pathtrav(n,l);
                    n := n.next;
                end loop
            end if
        end if
    end traverse
```

The pruning is particularly effective: it examines the minimum number of `child` and `next` pointers necessary to process the specified range. (Note however that this algorithm does *not* handle the case where any of the markers are negative.)

If the **process** procedure is defined to append nodes to a node list, then **pathtrav** is identical to the HyQ query `Pathloc(DOMTREE` *`range`* `TRUNC SET)`, i.e., it selects all document nodes (`DOMTREE`) located by *range*, implicitly truncating the range if it is out of bounds (`TRUNC`), and removing duplicate nodes (`SET`). (Strictly speaking, it does not "remove duplicate nodes"; rather, it only processes each node once to begin with.)

The **locate** function is a special-purpose form of the path traversal algorithm: it locates the *first* node in a given range. The **stepdown** function is an auxiliary routine which moves one level down the tree on the path from the root to a specified range.

```
function stepdown(n: nodePtr, l: pathloc) return nodePtr is
 begin
     if l.p + l.w <= n.p then
         return NULL;
     end if;
     n := n.child;
     while n /= NULL and n.p + n.w <= l.p loop
         n := n.next;
     end loop;
     return n;
 end stepdown
function locate(l : pathloc) return nodePtr is
     n : node := rootNode;
 begin
     while n /= NULL and n.d /= l.d loop
         n := stepdown(n,l);
     end loop;
     return n;
 end locate
```

Note that with the generalized binary tree representation, `pathloc` addresses are just as efficient as `treeloc` addresses for locating individual nodes, and are smaller on average.

Also, since **locate** and **stepdown** only examine the `path` and `depth` fields of the subtree position, only two markers are necessary to locate any individual tree node. (Cost 2 uses this technique to generate a compact string representation of node addresses.)

Range comparisons

Does one element contain another? `contains(x,y)` determines if `x` is a descendant of `y`, by examining their respective subtree positions.

```
function contains(x, y : nodePtr) return boolean is
     return x.p <= y.p and x.p + x.w >= b.p + b.w;
 end contains;
```

This algorithm works because of the tree properties. To compare two arbitrary path location addresses, it is necessary to examine the depth and height components as well:

```
function contains(q, r : pathloc) return boolean is
```

```
        return q.p <= r.p
            and q.d <= r.d
            and q.p + q.w >= r.p + r.w
            and q.d + q.h >= r.d + r.h;
    end contains;
```

Least common ancestor Two implementations of the least-common ancestor problem are given.

The first follows parent links of the deeper node until both pointers are at the same depth, then follows both pointers up the tree until the least common ancestor is found.

```
    function lca(x, y : nodePtr) return nodePtr is
    begin
        -- Note that this also works if x is an ancestor of y or vice versa
        while x.d < y.d loop x := x.parent; end loop;
        while y.d < x.d loop y := y.parent; end loop
        while x /= y loop
            x := x.parent;
            y := y.parent;
        end loop;
        return x;
    end lca;
```

The second implementation (which does not use parent links, and thus may be more suitable for other tree representations) works by stepping *down* the tree from the root until the paths to the two nodes diverge.

```
    function lca2(x, y :  nodePtr) return nodePtr is
        px, py, z : nodePtr;
    begin
        if contains(x,y) then return x; end if;
        if contains(y,x) then return y; end if;
        px := RootNode;
        py := RootNode;
        while px = py loop
            p := px;
            px := stepdown(p, (x.p, x.w, x.d, x.h));
            py := stepdown(p, (y.p, y.w, y.d, y.h));
        end loop
        return p;
    end lca2;
```

Locate all elements between two nodes The **between** function returns a path location address which selects all elements after node *x* and before node *y*, including *x*, excluding *y*, and including any ancestor nodes whose element boundaries are crossed.

```
    function between(x, y: nodePtr) return pathloc is
        z : nodePtr := lca(x,y);
    begin
        if before(y,x) then swap(x,y) end if;
```

```
        return (p => x.p, w => y.p − 1, d => z.d + 1, h => z.h − 1);
    end between;
```

Changing `y.p - 1` to `y.p + y.w` selects all nodes between *x* and *y* inclusive.

Relational locators All of the HyTime `relloc` forms can be expressed in terms of a root-relative path location address. While these relations can (with one exception) be processed just as easily by following next, parent, and child pointers, the pathloc equivalents are useful for illustration.

(The one exception is the `anc` relation: Using the pathloc traversal algorithm will select nodes in the "right" order for HyTime, i.e., from the root node down. This is a bit trickier to do by following pointers in the generalized binary tree representation.)

Some `relloc` forms take an additional dimspec argument which is interpreted as a *list location address* to be applied to the list of elements which satisfy the relation. The following routines do not handle this extra argument; it is assumed to be `1 -1` (meaning "the entire range").

```
function child(n : nodePtr) return pathloc is −− children
        return (p => n.p, w => n.w, d => n.d+1, h => 1);
    end;

    function anc(n : nodePtr) return pathloc is −− ancestors
        return (p => n.p, w => 1, d => 1, h => n.d);
    end

    function parent(n : nodePtr) return pathloc is
        return (p => n.p, w => 1, d => x.d−1, h => 1);
    end;

    function esib(n : nodePtr) return pathloc is −− elder or left siblings
        return (p => n.parent.p, w => n.p−n.parent.p, d => n.d, h => 1);
    end

    function ysib(n : nodePtr) return pathloc is −− younger or right siblings
        return (p => n.p + n.w, w => n.parent.p + n.parent.w − (n.p + n.w),
                d => n.d, h => 1);
    end;
```

The `des` or "descendants" relational locator form deserves special attention. The dimspec argument to this form is itself interpreted as a path location address. The **des** function below simply adjusts the argument so that it is relative to the document root instead of the subject node:

```
function des(n : nodePtr, arg : pathloc) return pathloc is
        return (p => x.p + arg.p − 1, w => arg.w,
                h => x.h + arg.h − 1, d => arg.d);
    end
```

Add: preceding (following); inputs: two nodes, subject and domain; returns: all descendants of domain node before (after) subject node

Need figure: show path list matrix, draw boxes around the ranges selected by each relloc form.

Breadth-first search A breadth-first search – process a node, then its direct children, then its children's children, and so on – is easy to implement in terms of **pathtrav**, by locating one row at a time.

```
procedure BFS(n : node) is
        range : pathloc;
  begin
        for i in n.d .. n.d + n.h loop
            range := (p => n.p, w => n.w, h => i, d => 1);
            pathtrav(n, range);
        end loop;
  end BFS;
```

Computing subtree locations

SGMLS and other parsers produce as output a stream of ESIS events, notifying the application for each start- and end-tag, data content, and so forth. (In fact, this is all that a parser is required to provide to a structure-controlled application.)

It is often necessary for an application to reconstruct the element hierarchy from the event stream. This can be accomplished in a straightforward manner my maintaining a stack of open elements and adding links as each event is received. It's possible to compute the subtree position of each node during this process, as illustrated by the following algorithm.

For clarity, a simplified version of the element structure information set is assumed, consisting only of start-tag, end-tag, and data events. Modifying this algorithm to handle the full set of ESIS data is straightforward, it just requires a *lot* of attention to detail. (See the Cost 2 source code, for example.)

```
type eventType is (startEvent, endEventEvent, dataEvent, eofEvent);

  declare
        ev : eventType;
        current : nodePtr;                  -- currently open node
        last : nodePtr;                     -- last closed node
        p : integerPtr;                     -- current path number
        n : nodePtr;
  begin
        rootNode := new node;
        current := rootNode
        current.next := current.child := current.parent := NULL;
        current.p := p := 1;
        current.d := 1;
        current.h := 1;
        last := NULL;

        ev := nextEvent();
```

```
    ASSERT ev = startEvent;          -- document element start-tag must be first
    ev := nextEvent();
    while ev /= eofEvent loop
        if ev = startEvent or ev = dataEvent then  -- Open new node
            n := new node;
            if last = NULL then              -- first child
                current.child := n;
            else                             -- insert at end
                last.next := n;
            end if;

            n.parent := current;
            n.next := NULL;
            n.child := NULL;
            n.p := p;
            -- width computed when node is closed
            n.d := current.d + 1;
            n.h := 1;                  -- height updated when processing children
            current := n;
            last := NULL;
            ev := nextEvent
        end if;

        if ev = endEventEvent or ev = dataEvent then     -- close current node
            if p = current.p then            -- this is a leaf node
                p := p + 1;
            end if
            current.w := p - current.p;
            if current.h >= current.parent.h then
                -- update height of parent node
                current.parent.h := current.h + 1
            end if
            last := current;
            current := current.parent;
        end if;
    end loop;
end;
```

For applications which modify the element structure dynamically, such as interactive editors and restructuring transformers, a different approach may be useful.

Since any modification to the tree changes the path number of *all* later nodes, it is better to store only the width and height fields of the path location address and compute the path number and depth on the fly. The depth can be computed by following `parent` links, and the path number can be computed (recursively) as the path number of the parent plus the sum of the widths of all earlier siblings. When nodes are inserted or deleted, changes to the width field must be propagated upward to all ancestor nodes.

What pathloc is *not* good for

Path location addresses are obviously not something that a human being will want to enter by hand. Also, fixed addresses hardcoded in HyQ queries like those shown in [3] are not likely to be very useful either unless the document structure is *very* rigid. As noted earlier, any change to the element structure can affect the path location addresses of all successive elements, so pathloc is not well-suited as an indexing mechanism in interactive editing environment.

However, due to its compactness and flexibility, pathloc could be quite useful for storing and transmitting the results of other queries.

References

Need proper references list

References

[1] DeRose, Steven J. and Durand, David G. Making Hypermedia Work: A User's Guide to HyTime Kluwer Academic Publishers, 1994. ISBN: 0-7923-9432-1.

[2] Ferris, Ralph E. and Newcomb, Victoria Taussig. HyTime Application Development Guide. Fujitsu Open Systems Solutions and TechnoTeacher, Inc. May 1995.

[3] Kimber, W. Eliot. HyTime and SGML: Understanding the HyTime HyQ Query Language. Technical report, Version 1.1. IBM Corporation, August 2, 1993.

[4] ISO/IEC DIS 1079.2. DSSSL, Document Style Semantics and Specification Language (draft).

Question: with `Pathloc(...  NOTSET)`*, in which order should nodes be returned? Column-wise or row-wise?*

Question: how are pathloc addresses with fourth marker (height) negative interpreted for "short" subtrees? In other words, what is the addressable range of the second dimspec when the query domain is a node whose leaves do not appear at the lowest level of all leaves in the document tree?