NAME

HTML::Tree::AboutTrees −− article on tree−shaped data structures in Perl

SYNOPSIS

```
# This an article, not a module.
```

DESCRIPTION

The following article by Sean M. Burke first appeared in *The Perl Journal* #18 and is copyright 2000 The Perl Journal. It appears courtesy of Jon Orwant and The Perl Journal. This document may be distributed under the same terms as Perl itself.

Trees

−− Sean M. Burke

> "AaaAAAaauugh! Watch out for that tree!"
> — *George of the Jungle theme*

Perl's facility with references, combined with its automatic management of memory allocation, makes it straightforward to write programs that store data in structures of arbitrary form and complexity.

But I've noticed that many programmers, especially those who started out with more restrictive languages, seem at home with complex but uniform data structures — N−dimensional arrays, or more struct-like things like hashes−of−arrays(−of−hashes(−of−hashes), etc.) — but they're often uneasy with building more freeform, less tabular structures, like tree-shaped data structures.

But trees are easy to build and manage in Perl, as I'll demonstrate by showing off how the HTML::Element class manages elements in an HTML document tree, and by walking you through a from-scratch implementation of game trees. But first we need to nail down what we mean by a "tree".

Socratic Dialogues: "What is a Tree?"

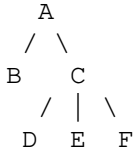
My first brush with tree-shaped structures was in linguistics classes, where tree diagrams are used to describe the syntax underlying natural language sentences. After learning my way around *those* trees, I started to wonder — are what I'm used to calling "trees" the same as what programmers call "trees"? So I asked lots of helpful and patient programmers how they would define a tree. Many replied with a answer in jargon that they could not really explain (understandable, since explaining things, especially defining things, is harder than people think):

> −− So what *is* a "tree", a tree-shaped data structure?
>
> −− A tree is a special case of an acyclic directed graph!
>
> −− What's a "graph"?
>
> −− Um... lines... and... you draw it... with... arcs! nodes! um...

The most helpful were folks who couldn't explain directly, but with whom I could get into a rather Socratic dialog (where *I* asked the half-dim half-earnest questions), often with much doodling of illustrations...

Question: so what's a tree?

Answer: A tree is a collection of nodes that are linked together in a, well, tree-like way! Like this *[drawing on a napkin]:*



Q: So what do these letters represent?

A: Each is a different node, a bunch of data. Maybe C is a bunch of data that stores a number, maybe a hash table, maybe nothing at all besides the fact that it links to D, E, and F (which are other nodes).

Q: So what're the lines between the nodes?

A: Links. Also called "arcs". They just symbolize the fact that each node holds a list of nodes it links to.

Q: So what if I draw nodes and links, like this...

```
  B -- E
 / \  / \
A   C
 \ /
  E
```

Is that still a tree?

A: No, not at all. There's a lot of un-treelike things about that. First off, E has a link coming off of it going into nowhere. You can't have a link to nothing — you can only link to another node. Second off, I don't know what that sideways link between B and E means...

Q: Okay, let's work our way up from something simpler. Is this a tree...?

```
A
```

A: Yes, I suppose. It's a tree of just one node.

Q: And how about...

```
A


B
```

A: No, you can't just have nodes floating there, unattached.

Q: Okay, I'll link A and B. How's this?

```
A
|
B
```

A: Yup, that's a tree. There's a node A, and a node B, and they're linked.

Q: How is that tree any different from this one...?

```
B
|
A
```

A: Well, in both cases A and B are linked. But it's in a different direction.

Q: Direction? What does the direction mean?

A: Well, it depends what the tree represents. If it represents a categorization, like this:

```
        citrus
      /    |    \
orange  lemon  kumquat ...
```

then you mean to say that oranges, lemons, kumquats, etc., are a kind of citrus. But if you drew it upside down, you'd be saying, falsely, that citrus is a kind of kumquat, a kind of lemon, and a kind of orange. If the tree represented cause-and-effect (or at least what situations could follow others), or represented what's a part of what, you wouldn't want to get those backwards, either. So with the nodes you draw together on paper, one has to be over the other, so you can tell which way the relationship in the tree works.

Q: So are these two trees the same?

```
  A           A
 / \         / \
B   C       B   \
                 C
```

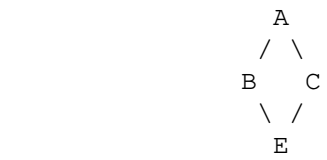
A: Yes, although by convention we often try to line up things in the same generation, like it is in the

diagram on the left.

Q: ‘‘generation’’? This is a family tree?

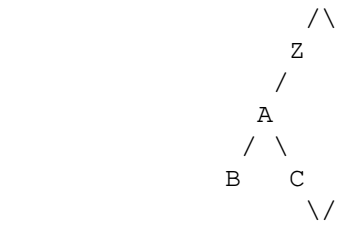
A: No, not unless it’s a family tree for just yeast cells or something else that reproduces asexually. But for sake of having lots of terms to use, we just pretend that links in the tree represent the ‘‘is a child of’’ relationship, instead of ‘‘is a kind of’’ or ‘‘is a part of’’, or ‘‘could result from’’, or whatever the real relationship is. So we get to borrow a lot of kinship words for describing trees — B and C are ‘‘children’’ (or ‘‘daughters’’) of A; A is the ‘‘parent’’ (or ‘‘mother’’) of B and C. Node C is a ‘‘sibling’’ (or ‘‘sister’’) of node C; and so on, with terms like ‘‘descendants’’ (a node’s children, children’s children, etc.), and ‘‘generation’’ (all the nodes at the same ‘‘level’’ in the tree, i.e., are either all grandchildren of the top node, or all great-grand-children, etc.), and ‘‘lineage’’ or ‘‘ancestors’’ (parents, and parent’s parents, etc., all the way to the topmost node).

So then we get to express rules in terms like "**A node cannot have more than one parent**", which means that this is not a valid tree:



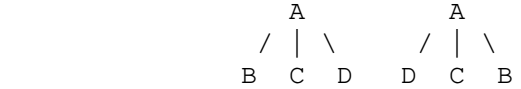
And: "**A node can’t be its own parent**", which excludes this looped-up connection:

Or, put more generally: "**A node can’t be its own ancestor**", which excludes the above loop, as well as the one here:

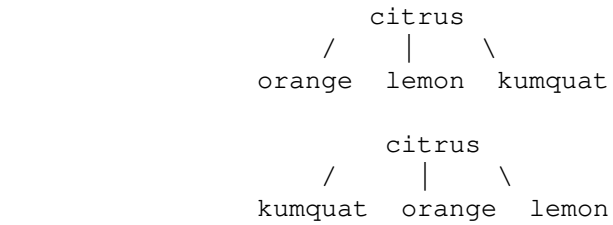


That tree is excluded because A is a child of Z, and Z is a child of C, and C is a child of A, which means A is its own great-grandparent. So this whole network can’t be a tree, because it breaks the sort of meta-rule: **once any node in the supposed tree breaks the rules for trees, you don’t have a tree anymore.**

Q: Okay, now, are these two trees the same?



A: It depends whether you’re basing your concept of trees on each node having a set (unordered list) of children, or an (ordered) list of children. It’s a question of whether ordering is important for what you’re doing. With my diagram of citrus types, ordering isn’t important, so these tree diagrams express the same thing:



because it doesn't make sense to say that oranges are "before" or "after" kumquats in the whole botanical scheme of things. (Unless, of course, you *are* using ordering to mean something, like a degree of genetic similarity.)

But consider a tree that's a diagram of what steps are comprised in an activity, to some degree of specificity:

```
           make tea
         /     |     \
   pour      infuse   serve
 hot water     / \
in cup/pot    /     \
            add     let
            tea     sit
           leaves
```

This means that making tea consists of putting hot water in a cup or put, infusing it (which itself consists of adding tea leaves and letting it sit), then serving it — *in that order*. If you serve an empty dry pot (sipping from empty cups, etc.), let it sit, add tea leaves, and pour in hot water, then what you're doing is performance art, not tea preparation:

```
         performance
             art
        /     |     \
   serve    infuse    pour
             / \       hot water
           /     \      in cup/pot
         let     add
         sit     tea
                leaves
```

Except for my having renamed the root, this tree is the same as the making-tea tree as far as what's under what, but it differs in order, and what the tree means makes the order important.

Q: Wait — "root"? What's a root?

A: Besides kinship terms like "mother" and "daughter", the jargon for tree parts also has terms from real-life tree parts: the part that everything else grows from is called the root; and nodes that don't have nodes attached to them (i.e., childless nodes) are called "leaves".

Q: But you've been drawing all your trees with the root at the top and leaves at the bottom.

A: Yes, but for some reason, that's the way everyone seems to think of trees. They can draw trees as above; or they can draw them sort of sideways with indenting representing what nodes are children of what:

```
* make tea
   * pour hot water in cup/pot
   * infuse
      * add tea leaves
      * let sit
   * serve
```

...but folks almost never seem to draw trees with the root at the bottom. So imagine it's based on spider plant in a hanging pot. Unfortunately, spider plants *aren't* botanically trees, they're plants; but "spider plant diagram" is rather a mouthful, so let's just call them trees.

Trees Defined Formally

In time, I digested all these assorted facts about programmers' ideas of trees (which turned out to be just a more general case of linguistic ideas of trees) into a single rule:

* A node is an item that contains ("is over", "is parent of", etc.) zero or more other nodes.

From this you can build up formal definitions for useful terms, like so:

* A node's **descendants** are defined as all its children, and all their children, and so on. Or, stated recursively: a node's descendants are all its children, and all its children's descendants. (And if it has no children, it has no descendants.)

* A node's **ancestors** consist of its parent, and its parent's parent, etc, up to the root. Or, recursively: a node's ancestors consist of its parent and its parent's ancestors. (If it has no parent, it has no ancestors.)

* A **tree** is a root node and all the root's descendants.

And you can add a proviso or two to clarify exactly what I impute to the word ''other'' in ''other nodes'':

* A node cannot contain itself, or contain any node that contains it, etc. Looking at it the other way: a node cannot be its own parent or ancestor.

* A node can be root (i.e., no other node contains it) or can be contained by only one parent; no node can be the child of two or more parents.

Add to this the idea that children are sometimes ordered, and sometimes not, and that's about all you need to know about defining what a tree is. From there it's a matter of using them.

Markup Language Trees: HTML-Tree

While not *all* markup languages are inherently tree-like, the best-known family of markup languages, HTML, SGML, and XML, are about as tree-like as you can get. In these languages, a document consists of elements and character data in a tree structure where there is one root element, and elements can contain either other elements, or character data.

> Footnote: For sake of simplicity, I'm glossing over comments (<!-- ... -->), processing instructions (<?xml version='1.0'>), and declarations (<!ELEMENT ...>, <!DOCTYPE ...>). And I'm not bothering to distinguish entity references (<, @) or CDATA sections (<![CDATA[...]]>) from normal text.

For example, consider this HTML document:

```
<html lang="en-US">
  <head>
    <title>
      Blank Document!
    </title>
  </head>
  <body bgcolor="#d010ff">
    I've got
    <em>
      something to saaaaay
    </em>
    !
  </body>
</html>
```

I've indented this to point out what nodes (elements or text items) are children of what, with each node on a line of its own.

The HTML::TreeBuilder module (in the CPAN distribution HTML-Tree) does the work of taking HTML source and building in memory the tree that the document source represents.

> Footnote: it requires the HTML::Parser module, which tokenizes the source — i.e., identifies each tag, bit of text, comment, etc.

The trees structures that it builds represent bits of text with normal Perl scalar string values; but elements are represented with objects — that is, chunks of data that belong to a class (in this case, HTML::Element), a class that provides methods (routines) for accessing the pieces of data in each element, and otherwise doing things with elements. (See my article in TPJ#17 for a quick explanation of objects, the POD document `perltoot` for a longer explanation, or Damian Conway's excellent book *Object-Oriented Perl* for the full story.)

Each HTML::Element object contains a number of pieces of data:

* its element name (“html”, “h1”, etc., accessed as `$element->tag`)

* a list of elements (or text segments) that it contains, if any (accessed as `$element->content_list` or `$element->content`, depending on whether you want a list, or an arrayref)

* what element, if any, contains it (accessed as `$element->parent`)

* and any SGML attributes that the element has, such as `lang="en-US"`, `align="center"`, etc. (accessed as `$element->attr('lang')`, `$element->attr('center')`, etc.)

So, for example, when HTML::TreeBuilder builds the tree for the above HTML document source, the object for the “body” element has these pieces of data:

```
* element name: "body"
* nodes it contains:
   the string "I've got "
   the object for the "em" element
   the string "!"
* its parent:
   the object for the "html" element
* bgcolor: "#d010ff"
```

Now, once you have this tree of objects, almost anything you'd want to do with it starts with searching the tree for some bit of information in some element.

Accessing a piece of information in, say, a hash of hashes of hashes, is straightforward:

```
$password{'sean'}{'sburke1'}{'hpux'}
```

because you know that all data points in that structure are accessible with that syntax, but with just different keys. Now, the “em” element in the above HTML tree does happen to be accessible as the root's child #1's child #1:

```
$root->content->[1]->content->[1]
```

But with trees, you typically don't know the exact location (via indexes) of the data you're looking for. Instead, finding what you want will typically involve searching through the tree, seeing if every node is the kind you want. Searching the whole tree is simple enough — look at a given node, and if it's not what you want, look at its children, and so on. HTML-Tree provides several methods that do this for you, such as `find_by_tag_name`, which returns the elements (or the first element, if called in scalar context) under a given node (typically the root) whose tag name is whatever you specify.

For example, that “em” node can be found as:

```
my $that_em = $root->find_by_tag_name('em');
```

or as:

```
@ems = $root->find_by_tag_name('em');
 # will only have one element for this particular tree
```

Now, given an HTML document of whatever structure and complexity, if you wanted to do something like change every

stuff

to

 [– *stuff* **–]**

the first step is to frame this operation in terms of what you're doing to the tree. You're changing this:

```
em
 |
...
```

to this:

```
      em
    /  |  \
   b  ...   b
   |        |
"[-"      "-]"
```

In other words, you're finding all elements whose tag name is ''em'', setting its class attribute to ''funky'', and adding one child to the start of its content list — a new ''b'' element whose content is the text string ''[-'' — and one to the end of its content list — a new ''b'' element whose content is the text string ''-]''.

Once you've got it in these terms, it's just a matter of running to the HTML::Element documentation, and coding this up with calls to the appropriate methods, like so:

```
use HTML::Element 1.53;
use HTML::TreeBuilder 2.96;
# Build the tree by parsing the document
my $root = HTML::TreeBuilder->new;
$root->parse_file('whatever.html'); # source file

# Now make new nodes where needed
foreach my $em ($root->find_by_tag_name('em')) {
  $em->attr('class', 'funky'); # Set that attribute

  # Make the two new B nodes
  my $new1 = HTML::Element->new('b');
  my $new2 = HTML::Element->new('b');
  # Give them content (they have none at first)
  $new1->push_content('[-');
  $new2->push_content('-]');

  # And put 'em in place!
  $em->unshift_content($new1);
  $em->push_content($new2);
}
print
 "<!-- Looky see what I did! -->\n",
 $root->as_HTML(), "\n";
```

The class HTML::Element provides just about every method I can image you needing, for manipulating trees made of HTML::Element objects. (And what it doesn't directly provide, it will give you the components to build it with.)

Building Your Own Trees

Theoretically, any tree is pretty much like any other tree, so you could use HTML::Element for anything you'd ever want to do with tree-arranged objects. However, as its name implies, HTML::Element is basically *for* HTML elements; it has lots of features that make sense only for HTML elements (like the idea that every element must have a tag-name). And it lacks some features that might be useful for general applications — such as any sort of checking to make sure that you're not trying to arrange objects in a non-treelike way. For a general-purpose tree class that does have such features, you can use Tree::DAG_Node, also available from CPAN.

However, if your task is simple enough, you might find it overkill to bother using Tree::DAG_Node. And, in any case, I find that the best way to learn how something works is to implement it (or something like it, but simpler) yourself. So I'll here discuss how you'd implement a tree structure, *without* using any of the existing classes for tree nodes.

Implementation: Game Trees for Alak

Suppose that the task at hand is to write a program that can play against a human opponent at a strategic board game (as opposed to a board game where there's an element of chance). For most such games, a "game tree" is an essential part of the program (as I will argue, below), and this will be our test case for implementing a tree structure from scratch.

For sake of simplicity, our game is not chess or backgammon, but instead a much simpler game called Alak. Alak was invented by the mathematician A. K. Dewdney, and described in his 1984 book *Planiverse*. The rules of Alak are simple:

> Footnote: Actually, I'm describing only my interpretation of the rules Dewdney describes in *Planiverse*. Many other interpretations are possible.

* Alak is a two-player game played on a one-dimensional board with eleven slots on it. Each slot can hold at most one piece at a time. There's two kinds of pieces, which I represent here as "x" and "o" — x's belong to one player (called X), o's to the other (called O).

* The initial configuration of the board is:

```
xxxx___oooo
```

For sake of the article, the slots are numbered from 1 (on the left) to 11 (on the right), and X always has the first move.

* The players take turns moving. At each turn, each player can move only one piece, once. (This unlike checkers, where you move one piece per move but get to keep moving it if you jump an your opponent's piece.) A player cannot pass up on his turn. A player can move any one of his pieces to the next unoccupied slot to its right or left, which may involve jumping over occupied slots. A player cannot move a piece off the side of the board.

* If a move creates a pattern where the opponent's pieces are surrounded, on both sides, by two pieces of the mover's color (with no intervening unoccupied blank slot), then those surrounded pieces are removed from the board.

* The goal of the game is to remove all of your opponent's pieces, at which point the game ends. Removing all-but-one ends the game as well, since the opponent can't surround you with one piece, and so will always lose within a few moves anyway.

Consider, then, this rather short game where X starts:

```
xxxx___oooo
  ^          Move 1: X moves from 3 (shown with caret) to 5
              (Note that any of X's pieces could move, but
              that the only place they could move to is 5.)
xx_xx__oooo
        ^    Move 2: O moves from 9 to 7.
xx_xx_oo_oo
   ^         Move 3: X moves from 4 to 6.
xx__xxoo_oo
         ^   Move 4: O (stupidly) moves from 10 to 9.
xx__xxooo_o
    ^        Move 5: X moves from 5 to 10, making the board
             "xx___xoooxo".  The three o's that X just
             surrounded are removed.
xx___x___xo
             O has only one piece, so has lost.
```

Now, move 4 could have gone quite the other way:

```
xx__xxoo_oo
            Move 4: O moves from 8 to 4, making the board
            "xx_oxxo__oo".  The surrounded x's are removed.
xx_o__o__oo
^           Move 5: X moves from 1 to 2.
_xxo__o__oo
      ^     Move 6: O moves from 7 to 6.
_xxo_o___oo
 ^          Move 7: X moves from 2 to 5, removing the o at 4.
__x_xo___oo
            ...and so on.
```

To teach a computer program to play Alak (as player X, say), it needs to be able to look at the configuration of the board, figure out what moves it can make, and weigh the benefit or costs, immediate or eventual, of those moves.

So consider the board from just before move 3, and figure all the possible moves X could make. X has pieces in slots 1, 2, 4, and 5. The leftmost two x's (at 1 and 2) are up against the end of the board, so they can move only right. The other two x's (at 4 and 5) can move either right or left:

```
Starting board: xx_xx_oo_oo
 moving 1 to 3 gives _xxxx_oo_oo
 moving 2 to 3 gives x_xxx_oo_oo
 moving 4 to 3 gives xxx_x_oo_oo
 moving 5 to 3 gives xxxx__oo_oo
 moving 4 to 6 gives xx__xxoo_oo
 moving 5 to 6 gives xx_x_xoo_oo
```

For the computer to decide which of these is the best move to make, it needs to quantify the benefit of these moves as a number — call that the "payoff". The payoff of a move can be figured as just the number of x pieces removed by the most recent move, minus the number of o pieces removed by the most recent move. (It so happens that the rules of the game mean that no move can delete both o's and x's, but the formula still applies.) Since none of these moves removed any pieces, all these moves have the same immediate payoff: 0.

Now, we could race ahead and write an Alak-playing program that could use the immediate payoff to decide which is the best move to make. And when there's more than one best move (as here, where all the moves are equally good), it could choose randomly between the good alternatives. This strategy is simple to implement; but it makes for a very dumb program. Consider what O's response to each of the potential moves (above) could be. Nothing immediately suggests itself for the first four possibilities (X having moved something to position 3), but either of the last two (illustrated below) are pretty perilous, because in either case O has the obvious option (which he would be foolish to pass up) of removing x's from the board:

```
 xx_xx_oo_oo
    ^       X moves 4 to 6.
 xx__xxoo_oo
        ^    O moves 8 to 4, giving "xx_oxxo__oo".  The two
             surrounded x's are removed.
 xx_o__o__oo
```

or

```
xx_xx_oo_oo
   ^       X moves 5 to 6.
xx_x_xoo_oo
       ^   O moves 8 to 5, giving "xx_xoxo__oo".  The one
           surrounded x is removed.
xx_xo_o__oo
```

Both contingencies are quite bad for X — but this is not captured by the fact that they start out with X thinking his move will be harmless, having a payoff of zero.

So what's needed is for X to think *more* than one step ahead — to consider not merely what it can do in this move, and what the payoff is, but to consider what O might do in response, and the payoff of those potential moves, and so on with X's possible responses to those cases could be. All these possibilities form a game tree — a tree where each node is a board, and its children are successors of that node — i.e., the boards that could result from every move possible, given the parent's board.

But how to represent the tree, and how to represent the nodes?

Well, consider that a node holds several pieces of data:

1) the configuration of the board, which, being nice and simple and one-dimensional, can be stored as just a string, like "xx_xx_oo_oo".

2) whose turn it is, X or O. (Or: who moved last, from which we can figure whose turn it is).

3) the successors (child nodes).

4) the immediate payoff of having moved to this board position from its predecessor (parent node).

5) and what move gets us from our predecessor node to here. (Granted, knowing the board configuration before and after the move, it's easy to figure out the move; but it's easier still to store it as one is figuring out a node's successors.)

6) whatever else we might want to add later.

These could be stored equally well in an array or in a hash, but it's my experience that hashes are best for cases where you have more than just two or three bits of data, or especially when you might need to add new bits of data. Moreover, hash key names are mnemonic — `$node->{'last_move_payoff'}` is plain as day, whereas it's not so easy having to remember with an array that `$node->[3]` is where you decided to keep the payoff.

> Footnote: Of course, there are ways around that problem: just swear you'll never use a real numeric index to access data in the array, and instead use constants with mnemonic names:
>
> ```
> use strict;
> use constant idx_PAYOFF => 3;
> ...
> $n->[idx_PAYOFF]
> ```
>
> Or use a pseudohash. But I prefer to keep it simple, and use a hash.
>
> These are, incidentally, the same arguments that people weigh when trying to decide whether their object-oriented modules should be based on blessed hashes, blessed arrays, or what. Essentially the only difference here is that we're not blessing our nodes or talking in terms of classes and methods.
>
> [end footnote]

So, we might as well represent nodes like so:

```
$node = { # hashref
   'board'          => ...board string, e.g., "xx_x_xoo_oo"

   'last_move_payoff' => ...payoff of the move
                         that got us here.
```

```
    'last_move_from' =>  ...the start...
    'last_move_to'   =>  ...and end point of the move
                             that got us here.  E.g., 5 and 6,
                             representing a move from 5 to 6.

    'whose_turn'     => ...whose move it then becomes.
                          just an 'x' or 'o'.

    'successors' => ...the successors
};
```

Note that we could have a field called something like 'last_move_who' to denote who last moved, but since turns in Alak always alternate (and no-one can pass), storing whose move it is now *and* who last moved is redundant — if X last moved, it's O turn now, and vice versa. I chose to have a 'whose_turn' field instead of a 'last_move_who', but it doesn't really matter. Either way, we'll end up inferring one from the other at several points in the program.

When we want to store the successors of a node, should we use an array or a hash? On the one hand, the successors to `$node` aren't essentially ordered, so there's no reason to use an array per se; on the other hand, if we used a hash, with successor nodes as values, we don't have anything particularly meaningful to use as keys. (And we can't use the successors themselves as keys, since the nodes are referred to by hash references, and you can't use a reference as a hash key.) Given no particularly compelling reason to do otherwise, I choose to just use an array to store all a node's successors, although the order is never actually used for anything:

```
$node = {
  ...
  'successors' => [ ...nodes... ],
  ...
};
```

In any case, now that we've settled on what should be in a node, let's make a little sample tree out of a few nodes and see what we can do with it:

```
# Board just before move 3 in above game
my $n0 = {
  'board' => 'xx_xx_oo_oo',
  'last_move_payoff' => 0,
  'last_move_from' =>  9,
  'last_move_to'   =>  7,
  'whose_turn' => 'x',
  'successors' => [],
};

# And, for now, just two of the successors:

# X moves 4 to 6, giving xx__xxoo_oo
my $n1 = {
  'board' => 'xx__xxoo_oo',
  'last_move_payoff' => 0,
  'last_move_from' =>  4,
  'last_move_to'   =>  6,
  'whose_turn' => 'o',
  'successors' => [],
};

# or X moves 5 to 6, giving xx_x_xoo_oo
```

```
my $n2 = {
  'board' => 'xx_x_xoo_oo',
  'last_move_payoff' => 0,
  'last_move_from' =>  5,
  'last_move_to'   =>  6,
  'whose_turn' => 'o',
  'successors' => [],
};

# Now connect them...
push @{$n0->{'successors'}}, $n1, $n2;
```

Digression: Links to Parents

In comparing what we store in an Alak game tree node to what HTML::Element stores in HTML element nodes, you'll note one big difference: every HTML::Element node contains a link to its parent, whereas we don't have our Alak nodes keeping a link to theirs.

The reason this can be an important difference is because it can affect how Perl knows when you're not using pieces of memory anymore. Consider the tree we just built, above:

```
     node 0
    /      \
node 1    node 2
```

There's two ways Perl knows you're using a piece of memory: 1) it's memory that belongs directly to a variable (i.e., is necessary to hold that variable's value, or value*s* in the case of a hash or array), or 2) it's a piece of memory that something holds a reference to. In the above code, Perl knows that the hash for node 0 (for board "xx_xx_oo_oo") is in use because something (namely, the variable `$n0`) holds a reference to it. Now, even if you followed the above code with this:

```
$n1 = $n2 = 'whatever';
```

to make your variables `$n1` and `$n2` stop holding references to the hashes for the two successors of node 0, Perl would still know that those hashes are still in use, because node 0's successors array holds a reference to those hashes. And Perl knows that node 0 is still in use because something still holds a reference to it. Now, if you added:

```
my $root = $n0;
```

This would change nothing — there's just be *two* things holding a reference to the node 0 hash, which in turn holds a reference to the node 1 and node 2 hashes. And if you then added:

```
$n0 = 'stuff';
```

still nothing would change, because something (`$root`) still holds a reference to the node 0 hash. But once *nothing* holds a reference to the node 0 hash, Perl will know it can destroy that hash (and reclaim the memory for later use, say), and once it does that, nothing will hold a reference to the node 1 or the node 2 hashes, and those will be destroyed too.

But consider if the node 1 and node 2 hashes each had an attribute "parent" (or "predecessor") that held a reference to node 0. If your program stopped holding a reference to the node 0 hash, Perl could *not* then say that *nothing* holds a reference to node 0 — because node 1 and node 2 still do. So, the memory for nodes 0, 1, and 2 would never get reclaimed (until your program ended, at which point Perl destroys *everything*). If your program grew and discarded lots of nodes in the game tree, but didn't let Perl know it could reclaim their memory, your program could grow to use immense amounts of memory — never a nice thing to have happen. There's three ways around this:

1) When you're finished with a node, delete the reference each of its children have to it (in this case, deleting `$n1->{'parent'}`, say). When you're finished with a whole tree, just go through the whole tree erasing links that children have to their children.

2) Reconsider whether you really need to have each node hold a reference to its parent. Just not having

those links will avoid the whole problem.

3) use the WeakRef module with Perl 5.6 or later. This allows you to "weaken" some references (like the references that node 1 and 2 could hold to their parent) so that they don't count when Perl goes asking whether anything holds a reference to a given piece of memory. This wonderful new module eliminates the headaches that can often crop up with either of the two previous methods.

It so happens that our Alak program is simple enough that we don't need for our nodes to have links to their parents, so the second solution is fine. But in a more advanced program, the first or third solutions might be unavoidable.

Recursively Printing the Tree

I don't like working blind — if I have any kind of a complex data structure in memory for a program I'm working on, the first thing I do is write something that can dump that structure to the screen so I can make sure that what I *think* is in memory really *is* what's in memory. Now, I could just use the "x" pretty-printer command in Perl's interactive debugger, or I could have the program use the `Data::Dumper` module. But in this case, I think the output from those is rather too verbose. Once we have trees with dozens of nodes in them, we'll really want a dump of the tree to be as concise as possible, hopefully just one line per node. What I'd like is something that can print `$n0` and its successors (see above) as something like:

```
xx_xx_oo_oo  (O moved 9 to 7, 0 payoff)
  xx__xxoo_oo  (X moved 4 to 6, 0 payoff)
  xx_x_xoo_oo  (X moved 5 to 6, 0 payoff)
```

A subroutine to print a line for a given node, and then do that again for each successor, would look something like:

```
sub dump_tree {
  my $n = $_[0]; # "n" is for node
  print
    ...something expressing $n'n content...
  foreach my $s (@{$n->{'successors'}}) {
    # "s for successor
    dump($s);
  }
}
```

And we could just start that out with a call to `dump_tree($n0)`.

Since this routine...

> Footnote: I first wrote this routine starting out with "sub dump {". But when I tried actually calling `dump($n0)`, Perl would dump core! Imagine my shock when I discovered that this is absolutely to be expected — Perl provides a built-in function called `dump`, the purpose of which is to, yes, make Perl dump core. Calling our routine "dump_tree" instead of "dump" neatly avoids that problem.

...does its work (dumping the subtree at and under the given node) by calling itself, it's **recursive**. However, there's a special term for this kind of recursion across a tree: traversal. To **traverse** a tree means to do something to a node, and to traverse its children. There's two prototypical ways to do this, depending on what happens when:

```
traversing X in pre-order:
  * do something to X
  * then traverse X's children

traversing X in post-order:
  * traverse X's children
  * then do something to X
```

Dumping the tree to the screen the way we want it happens to be a matter of pre-order traversal, since the thing we do (print a description of the node) happens before we recurse into the successors.

When we try writing the `print` statement for our above `dump_tree`, we can get something like:

```
sub dump_tree {
  my $n = $_[0];

  # "xx_xx_oo_oo  (O moved 9 to 7, 0 payoff)"
  print
    $n->{'board'}, "  (",
    ($n->{'whose_turn'} eq 'o' ? 'X' : 'O'),
    # Infer who last moved from whose turn it is now.
    " moved ", $n->{'last_move_from'},
    " to ",    $n->{'last_move_to'},
    ", ",      $n->{'last_move_payoff'},
    " payoff)\n",
  ;

  foreach my $s (@{$n->{'successors'}}) {
    dump_tree($s);
  }
}
```

If we run this on `$n0` from above, we get this:

```
xx_xx_oo_oo  (O moved 9 to 7, 0 payoff)
xx__xxoo_oo  (X moved 4 to 6, 0 payoff)
xx_x_xoo_oo  (X moved 5 to 6, 0 payoff)
```

Each line on its own is fine, but we forget to allow for indenting, and without that we can't tell what's a child of what. (Imagine if the first successor had successors of its own — you wouldn't be able to tell if it were a child, or a sibling.) To get indenting, we'll need to have the instances of the `dump_tree` routine know how far down in the tree they're being called, by passing a depth parameter between them:

```
sub dump_tree {
  my $n = $_[0];
  my $depth = $_[1];
  $depth = 0 unless defined $depth;
  print
    "  " x $depth,
    ...stuff...
  foreach my $s (@{$n->{'successors'}}) {
    dump_tree($s, $depth + 1);
  }
}
```

When we call `dump_tree($n0)`, `$depth` (from `$_[1]`) is undefined, so gets set to 0, which translates into an indenting of no spaces. But when `dump_tree` invokes itself on `$n0`'s children, those instances see `$depth + 1` as their `$_[1]`, giving appropriate indenting.

Footnote: Passing values around between different invocations of a recursive routine, as shown, is a decent way to share the data. Another way to share the data is by keeping it in a global variable, like `$Depth`, initially set to 0. Each time `dump_tree` is about to recurse, it must `++$Depth`, and when it's back, it must `--$Depth`.

Or, if the reader is familiar with closures, consider this approach:

```
sub dump_tree {
  # A wrapper around calls to a recursive closure:
  my $start_node = $_[0];
  my $depth = 0;
   # to be shared across calls to $recursor.
  my $recursor;
  $recursor = sub {
    my $n = $_[0];
    print "  " x $depth,
      ...stuff...
    ++$depth;
    foreach my $s (@{$n->{'successors'}}) {
      $recursor->($s);
    }
    --$depth;
  }
  $recursor->($start_node); # start recursing
  undef $recursor;
}
```

The reader with an advanced understanding of Perl's reference-count-based garbage collection is invited to consider why it is currently necessary to undef `$recursor` (or otherwise change its value) after all recursion is done.

The reader whose mind is perverse in other ways is invited to consider how (or when!) passing a depth parameter around is unnecessary because of information that Perl's `caller(N)` function reports!

[end footnote]

Growing the Tree

Our `dump_tree` routine works fine for the sample tree we've got, so now we should get the program working on making its own trees, starting from a given board.

In `Games::Alak` (the CPAN-released version of Alak that uses essentially the same code that we're currently discussing the tree-related parts of), there is a routine called `figure_successors` that, given one childless node, will figure out all its possible successors. That is, it looks at the current board, looks at every piece belonging to the player whose turn it is, and considers the effect of moving each piece every possible way — notably, it figures out the immediate payoff, and if that move would end the game, it notes that by setting an "endgame" entry in that node's hash. (That way, we know that that's a node that *can't* have successors.)

In the code for `Games::Alak`, `figure_successors` does all these things, in a rather straightforward way. I won't walk you through the details of the `figure_successors` code I've written, since the code has nothing much to do with trees, and is all just implementation of the Alak rules for what can move where, with what result. Especially interested readers can puzzle over that part of code in the source listing in the archive from CPAN, but others can just assume that it works as described above.

But consider that `figure_successors`, regardless of its inner workings, does not grow the *tree*; it only makes one set of successors for one node at a time. It has to be up to a different routine to call `figure_successors`, and to keep applying it as needed, in order to make a nice big tree that our game-playing program can base its decisions on.

Now, we could do this by just starting from one node, applying `figure_successors` to it, then applying `figure_successors` on all the resulting children, and so on:

```
sub grow {  # Just a first attempt at this!
  my $n = $_[0];
  figure_successors($n);
   unless
    @{$n->{'successors'}}
      # already has successors.
    or $n->{'endgame'}
      # can't have successors.
  }
  foreach my $s (@{$n->{'successors'}}) {
    grow($s); # recurse
  }
}
```

If you have a game tree for tic-tac-toe, and you grow it without limitation (as above), you will soon enough have a fully "solved" tree, where every node that *can* have successors *does*, and all the leaves of the tree are *all* the possible endgames (where, in each case, the board is filled). But a game of Alak is different from tic-tac-toe, because it can, in theory, go on forever. For example, the following sequence of moves is quite possible:

```
xxxx___oooo
xxx_x__oooo
xxx_x_o_ooo
xxxx__o_ooo (x moved back)
xxxx___oooo (o moved back)
...repeat forever...
```

So if you tried using our above attempt at a `grow` routine, Perl would happily start trying to construct an infinitely deep tree, containing an infinite number of nodes, consuming an infinite amount of memory, and requiring an infinite amount of time. As the old saying goes: "You can't have everything — where would you put it?" So we have to place limits on how much we'll grow the tree.

There's more than one way to do this:

1. We could grow the tree until we hit some limit on the number of nodes we'll allow in the tree.

2. We could grow the tree until we hit some limit on the amount of time we're willing to spend.

3. Or we could grow the tree until it is fully fleshed out to a certain depth.

Since we already know to track depth (as we did in writing `dump_tree`), we'll do it that way, the third way. The implementation for that third approach is also pretty straightforward:

```
$Max_depth = 3;
sub grow {
  my $n = $_[0];
  my $depth = $_[1] || 0;
  figure_successors($n)
   unless
    $depth >= $Max_depth
    or @{$n->{'successors'}}
    or $n->{'endgame'}
  }
  foreach my $s (@{$n->{'successors'}}) {
    grow($s, $depth + 1);
  }
  # If we're at $Max_depth, then figure_successors
  #  didn't get called, so there's no successors
  #  to recurse under -- that's what stops recursion.
}
```

If we start from a single node (whether it's a node for the starting board "xxxx___oooo", or for whatever board the computer is faced with), set `$Max_depth` to 4, and apply `grow` to it, it will grow the tree to include several hundred nodes.

> Footnote: If at each move there are four pieces that can move, and they can each move right or left, the "branching factor" of the tree is eight, giving a tree with 1 (depth 0) + 8 (depth 1) + 8 ** 2 + 8 ** 3 + 8 ** 4 = 4681 nodes in it. But, in practice, not all pieces can move in both directions (none of the x pieces in "xxxx___oooo" can move left, for example), and there may be fewer than four pieces, if some were lost. For example, there are 801 nodes in a tree of depth four starting from "xxxx___oooo", suggesting an average branching factor of about five (801 ** (1/4) is about 5.3), not eight.

What we need to derive from that tree is the information about what are the best moves for X. The simplest way to consider the payoff of different successors is to just average them — but what we average isn't always their immediate payoffs (because that'd leave us using only one generation of information), but the average payoff of *their* successors, if any. We can formalize this as:

```
To figure a node's average payoff:
  If the node has successors:
    Figure each successor's average payoff.
    My average payoff is the average of theirs.
  Otherwise:
    My average payoff is my immediate payoff.
```

Since this involves recursing into the successors *before* doing anything with the current node, this will traverse the tree *in post-order*.

We could work that up as a routine of its own, and apply that to the tree after we've applied `grow` to it. But since we'd never grow the tree without also figuring the average benefit, we might as well make that figuring part of the `grow` routine itself:

```
$Max_depth = 3;
sub grow {
  my $n = $_[0];
  my $depth = $_[1] || 0;
  figure_successors($n);
   unless
    $depth >= $Max_depth
    or @{$n->{'successors'}}
    or $n->{'endgame'}
  }

  if(@{$n->{'successors'}}) {
    my $a_payoff_sum = 0;
    foreach my $s (@{$n->{'successors'}}) {
      grow($s, $depth + 1);  # RECURSE
      $a_payoff_sum += $s->{'average_payoff'};
    }
    $n->{'average_payoff'}
     = $a_payoff_sum / @{$n->{'successors'}};
  } else {
    $n->{'average_payoff'}
     = $n->{'last_move_payoff'};
  }
}
```

So, by time `grow` has applied to a node (wherever in the tree it is), it will have figured successors if possible (which, in turn, sets `last_move_payoff` for each node it creates), and will have set `average_benefit`.

Beyond this, all that's needed is to start the board out with a root note of "xxxx___oooo", and have the computer (X) take turns with the user (O) until someone wins. Whenever it's O's turn, `Games::Alak` presents a prompt to the user, letting him know the state of the current board, and asking what move he selects. When it's X's turn, the computer grows the game tree as necessary (using just the `grow` routine from above), then selects the move with the highest average payoff (or one of the highest, in case of a tie).

In either case, "selecting" a move means just setting that move's node as the new root of the program's game tree. Its sibling nodes and their descendants (the boards that *didn't* get selected) and its parent node will be erased from memory, since they will no longer be in use (as Perl can tell by the fact that nothing holds references to them anymore).

The interface code in `Games::Alak` (the code that prompts the user for his move) actually supports quite a few options besides just moving — including dumping the game tree to a specified depth (using a slightly fancier version of `dump_tree`, above), resetting the game, changing `$Max_depth` in the middle of the game, and quitting the game. Like `figure_successors`, it's a bit too long to print here, but interested users are welcome to peruse (and freely modify) the code, as well as to enjoy just playing the game.

Now, in practice, there's more to game trees than this: for games with a larger branching factor than Alak has (which is most!), game trees of depth four or larger would contain too many nodes to be manageable, most of those nodes being strategically quite uninteresting for either player; dealing with game trees specifically is therefore a matter of recognizing uninteresting contingencies and not bothering to grow the tree under them.

> Footnote: For example, to choose a straightforward case: if O has a choice between moves that put him in immediate danger of X winning and moves that don't, then O won't ever choose the dangerous moves (and if he does, the computer will know enough to end the game), so there's no point in growing the tree any further beneath those nodes.

But this sample implementation should illustrate the basics of how to build and manipulate a simple tree structure in memory. And once you've understood the basics of tree storage here, you should be ready to better understand the complexities and peculiarities of other systems for creating, accessing, and changing trees, including Tree::DAG_Node, HTML::Element, XML::DOM, or related formalisms like XPath and XSL.

[end body of article]

[Author Credit]

Sean M. Burke (`sburke@cpan.org`) is a tree-dwelling hominid.

References

Dewdney, A[lexander] K[eewatin]. 1984. *Planiverse: Computer Contact with a Two-Dimensional World.* Poseidon Press, New York.

Knuth, Donald Ervin. 1997. *Art of Computer Programming, Volume 1, Third Edition: Fundamental Algorithms*. Addison-Wesley, Reading, MA.

Wirth, Niklaus. 1976. *Algorithms + Data Structures = Programs* Prentice-Hall, Englewood Cliffs, NJ.

Worth, Stan and Allman Sheldon. Circa 1967. *George of the Jungle* theme. [music by Jay Ward.]

Wirth's classic, currently and lamentably out of print, has a good section on trees. I find it clearer than Knuth's (if not quite as encyclopedic), probably because Wirth's example code is in a block-structured high-level language (basically Pascal), instead of in assembler (MIX). I believe the book was re-issued in the 1980s under the titles *Algorithms and Data Structures* and, in a German edition, *Algorithmen und Datenstrukturen*. Cheap copies of these editions should be available through used book services such as `abebooks.com`.

Worth's classic, however, is available on the soundtrack to the 1997 *George of the Jungle* movie, as performed by The Presidents of the United States of America.

BACK

Return to the HTML::Tree docs.