Jeremy Bowers

15th November 2002

As part of re-doing the Navigation Maps screen for LON-CAPA, I created an iterator for traversing the maps. Since I'm not going to be around forever, I wanted to document the most subtle part of the the algorithm I developed, so later people don't need to read my mind. Since I wanted to use pictures, that rules out trying to do it all in comments.

In particular, the handling of branches is a challenge. So, suppose we have the simple navigation map shown in figure 1. As human beings, we can look at that resource map and mentally represent that as the following: *Everybody sees resource A, then you have a choice of resources B and C, or resources E and F, and everybody will do resource D.* We want to represent that like this:

- A
  - B or C
  - E or F
- D

Or something like that. Actually, LaTeX fails me here. At the very least, I don't want a bullet in front of D, since that follows A, and in the real nav maps, "B or C" is represented in two rows, but you get the idea.

The problem is that the computer doesn't have the global, overall view we get, and getting the computer to match our intuition is surprisingly difficult.

If we consider the *desired indentation depth* (or just *depth* for short) as what we are looking for, so we can display the navmaps correctly, then we are looking for two things:

1. A mapping of the nodes to some depth.
2. A way of traversing the graph once the depth has been assigned.

1 Mapping The Nodes to Some Depth

As I talk about the algorithm, markers like ****1**** will be used to track the places in the source code that are being discussed.

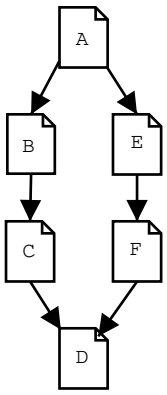

Figure 1: Simple Branching Nav Map

Mapping the nodes to a depth is done via a two-pass algorithm (**1**). The passes are virtually identical, but one starts from the top and one starts from the bottom.

The order the nodes are traversed in is not terribly importent, as long as no node is encountered before one of its parent nodes have been encountered, which is known to be true with this traversal (and most others, as well).

For the top-down pass, we start with the first resource, and give it a "top-down-value" (TDV for short) value of 0 (**2**). We then progress along the links in some order, visiting resources and assigning them a value as follows:

1. If the node we came from has only one link leaving it (i.e., this is the only next node for that node), then this node is given the same TDV value as the parent. (**3**)

2. If the node we came from has more then one link leaving it, this node is given the TDV value of the parent, plus one. (**4**)

3. If we get to any given node through multiple paths, the minimal value is taken. (See all uses of the "min" function.)

Using that algorithm on the given example navmap, we get the result shown in figure 2. **A** was given the value 0 to start with. Each of **B** and **E** was given 1, because there are two ways out of **A**. The rest of the resources were given the value 1 as well, because there is only one way out of the resources from that point on.

Using the same algorithm on the link-reversed graph, starting at the finish resource, we can do the same thing to get the "Bottom-Up-Values" (BUV). We can then take each node, and take the minimum of each value to get the final Depth (D) value.. This is shown in figure 3.

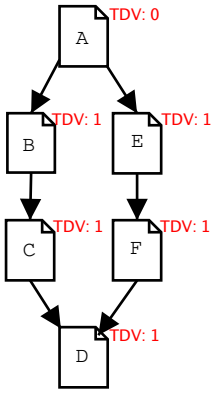

Figure 2: TDV Values

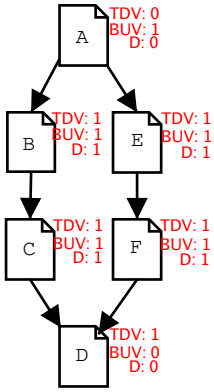

Figure 3: Final Depth labels

2 Traversing the graph

Now that we've assigned desired depths, we need to figure out an order for traversing the graph. I create a list of arrays, of the size of the maximum depth, which I use to keep track of the links we need to keep track of. This is stored in the variable {STACK}, which holds an array reference of array references of that size.

In the shown example, the maximum depth is two, so $self->{STACK} will be [[], []]. **A** and **D** will eventually end up in the first array, **B**, **C**, **E**, and **F** will be in the second.

Basically, the procedure is this:

- Select the next resource to display.
- Explore where that resource can get us, and store it in the stack.
- Repeat until there are no more resources.

We start by priming the stack with the first resource to be examined, the firstResource (****5****).

In order to select the next resource to display, we walk backwards along the stack until we find the first non-empty list (****6****). We pop the resource off that list, and that is the next resource (****7****). This has the effect of following branches as quickly as possible, so that we can guarentee that **B**, **C**, and **E**, **F** will both be explored before **D**. **D** will not be selected until the higher levels have been exhausted.

Of course this works recursively, so it will work no matter how high the level goes.

The only thing that needs a special case is when there are two branches, as in the sample figure, and we get to the end of the first branch (****8****). We can't detect the fact that the branch has ended just by looking at the stack afterwards. So to remember that a branch has ended, detected when all the potential next resources are of a lower level then the current resource, even if there are no possible next resources, we push a marker onto the corresponding stack that the branch has ended (****9****). We have to push a marker onto the stack, because we may still need to recursively explore the last resource if it's a map.

And that's pretty much it; there's a lot of bookkeeping and stuff, but that's the core.