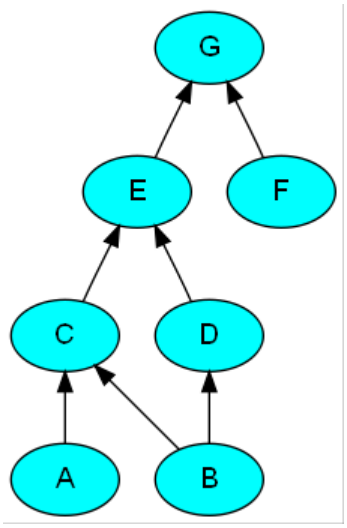
Finding Directed Acyclic Graph (DAG) Minimal Elements (Vertices) with XSLT/XPath?

I have an XML file that encodes a directed acyclic graph (DAG) that represents a partial order. Such graphs are useful for things like specifying dependencies and finding critical paths. For the curious, my current application is to specify component dependencies for a build system, so vertices are components and edges specify compile-time dependencies. Here is a simple example:

```
<?xml version="1.0"?>
<dag>
    <vertex name="A">
        <directed-edge-to vertex="C"/>
    </vertex>
    <vertex name="B">
        <directed-edge-to vertex="C"/>
        <directed-edge-to vertex="D"/>
    </vertex>
    <vertex name="C">
        <directed-edge-to vertex="E"/>
    </vertex>
    <vertex name="D">
        <directed-edge-to vertex="E"/>
    </vertex>
    <vertex name="E">
        <directed-edge-to vertex="G"/>
    </vertex>
    <vertex name="F">
        <directed-edge-to vertex="G"/>
    </vertex>
    <vertex name="G"/>
</dag>
```

This DAG may be drawn like this:



I'd like to apply an XSLT stylesheet that produces another XML document that contains only the vertices that correspond to *minimal elements* of the partial order. That is, those vertices that have no incoming edges. The set of minimal vertices for the example graph is `{A, B, F}`. For my build dependency application, finding this set is valuable because I know that if I build the members of this set, then everything in my project will be built.

Here is my current stylesheet solution (I'm running this with Xalan on Java using Apache Ant's `xslt` task). A key observation is that a minimal vertex will not be referenced in any `directed-edge-to` element:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:xalan="http://xml.apache.org/xslt"
                exclude-result-prefixes="xalan">
    <xsl:output method="xml" indent="yes" xalan:indent-amount="4"/>

    <xsl:template match="dag">
        <minimal-vertices>
            <xsl:for-each select="//vertex">
                <xsl:if test="not(//vertex/directed-edge-to[@vertex=current()/@nam
                    <minimal-vertex name="{@name}"/>
                </xsl:if>
            </xsl:for-each>
        </minimal-vertices>
    </xsl:template>
</xsl:stylesheet>
```

Applying this stylesheet produces the following output (which I believe is correct):

```
<?xml version="1.0" encoding="UTF-8"?>
<minimal-vertices>
    <minimal-vertex name="A"/>
    <minimal-vertex name="B"/>
    <minimal-vertex name="F"/>
</minimal-vertices>
```

The thing is, I'm not completely satisfied with this solution. **I'm wondering if there is a way to combine the `select` of the `for-each` and the `test` of the `if` with XPath syntax.**

I want to write something like:

```
<xsl:for-each select="//vertex[not(//vertex/directed-edge-to[@vertex=current()/@na
```

But that does not do what I want because the `current()` function does not reference the nodes selected by the outer `//vertex` expression.

Thusfar, my solution uses XPath 1.0 and XSLT 1.0 syntax, though I'm open to XPath 2.0 and XSLT 2.0 syntax as well.

Here's the Ant build script if you like:

```
<?xml version="1.0"?>
<project name="minimal-dag" default="default">
    <target name="default">
        <xslt in="dag.xml" out="minimal-vertices.xml" style="find-minimal-vertices
    </target>
    <target name="dot">
        <xslt in="dag.xml" out="dag.dot" style="xml-to-dot.xsl"/>
    </target>
</project>
```

The `dot` target generates Graphviz Dot language code for rendering the graph. Here is `xml-to-dot.xsl`:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:xalan="http://xml.apache.org/xslt"
                exclude-result-prefixes="xalan">
    <xsl:output method="text"/>

    <xsl:template match="dag">
        digraph {
        rankdir="BT";
        node [style="filled", fillcolor="cyan", fontname="Helvetica"];
        <xsl:apply-templates select="//directed-edge-to"/>
        }
    </xsl:template>

    <xsl:template match="directed-edge-to">
        <xsl:value-of select="concat(ancestor::vertex/@name, '->', @vertex, ';')"/
    </xsl:template>
</xsl:stylesheet>
```

xslt | xpath | graph-theory

edited **Dec 8 '11 at 18:49**

asked **May 9 '09 at 19:28**

Greg Mattes
4,254 1 17 43
50% accept rate

1 The "//" abbreviation should be avoided whenever possible as it is very expensive, causing the whole subtree rooted at the context node to be searched. "//" at the top level causes the whole XML document to be searched. It is very important *not* to use "//" whenever the structure of the XML document is known at the time of writing the XPath expression – Dimitre Novatchev May 11 '09 at 16:44

feedback

2 Answers

You can take advantage of XPath's implicit existential quantification on the `=` operator:

```
<xsl:for-each select="//vertex[not(@name = //vertex/directed-edge-to/@vertex)]">
```

When you use any of the six comparison operators (`=` , `!=` , `<` , `<=` , `>` , and `>=`) to compare a node-set, the expression will return true if any node in the node-set satisfies the condition. When comparing one node-set with another, the expression returns true if any node in the first node-set satisfies the condition when compared with any node in the second node-set. XPath 2.0 introduces six new operators that don't perform this existential quantification (`eq` , `ne` , `lt` , `le` , `gt` , and `ge`). But in your case, you'll want to use " = " to get that existential quantification.

Note of course, that you'll still want to use the `not()` function as you were doing. Most of the time, it's good to avoid the `!=` operator. If you used it here instead of `not()` , then it would return true if there are any `@vertex` attributes that are not equal to the `@name` value, which is not your intention. (And if either node-set is empty, then it would return false, as comparisons with empty node-sets always return false.)

If you want to use `eq` instead, then you'd have to do something like you did: separate out the conditional from the iteration so you could bind `current()` . But in XPath 2.0, you can do this within an expression:

```
<xsl:for-each select="for $v in //vertex
                      return $v[not(//directed-edge-to[@vertex eq $v/@name])]">
```

This is useful for when your condition isn't a simple equality comparison (and thus can't be existentially quantified using " = "). For example: `starts-with(@vertex, $v/@name)` .

XPath 2.0 also has an explicit way of performing existential quantification. Instead of the `for` expression above, we could have written this:

```
<xsl:for-each select="//vertex[not(some $e in //directed-edge-to
                                   satisfies @name eq $e/@vertex)]">
```

In addition to the " `some` " syntax, XPath 2.0 also supplies a corresponding " `every` " syntax for performing *universal* quantification.

Rather than using `for-each`, you could also use template rules, which are more modular (and powerful):

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    <minimal-vertices>
      <xsl:apply-templates/>
    </minimal-vertices>
  </xsl:template>

  <!-- Copy vertex elements that have no arrows pointing to them -->
  <xsl:template match="vertex[not(@name = //directed-edge-to/@vertex)]">
    <minimal-vertex name="{@name}"/>
  </xsl:template>

</xsl:stylesheet>
```

Again, in this case, we're relying on the existential quantification of `=`.

XSLT 1.0 prohibits use of the `current()` function in patterns, i.e., in the `match` attribute, but XSLT 2.0 allows it. In that case, `current()` refers to the node currently being matched. So in XSLT 2.0, we could also write this (without having to use a `for` expression):

```
<xsl:template match="vertex[not(//directed-edge-to[@vertex eq current()/@name])]">
```

Note that this pattern is essentially the same as the expression you tried to use in `for-each`, but whereas it doesn't do what you want in `for-each`, it *does* do what you want in the pattern (because what `current()` binds to is different).

Finally, I'll add one more variation that in some ways simplifies the logic (removing `not()`). This also goes back to using XSLT 1.0:

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    <minimal-vertices>
      <xsl:apply-templates/>
    </minimal-vertices>
  </xsl:template>

  <!-- By default, copy vertex elements -->
  <xsl:template match="vertex">
    <minimal-vertex name="{@name}"/>
  </xsl:template>

  <!-- But strip out vertices with incoming arrows -->
  <xsl:template match="vertex[@name = //directed-edge-to/@vertex]"/>

</xsl:stylesheet>
```

If you don't like the whitespace being output, add an empty rule for text nodes, so they'll get stripped out (overriding the default rule for text nodes, which is to copy them):

```
<xsl:template match="text()"/>
```

Or you could just be more selective in what nodes you apply templates to:

```
<xsl:apply-templates select="/dag/vertex"/>
```

Which approach you take is partially dependent on taste, partially dependent on the wider context of your stylesheet and expected data (how much the input structure might vary, etc.).

I know I went way beyond what you were asking for, but I hope you at least found this interesting. :-)

edited **May 10 '09 at 10:58** answered **May 10 '09 at 10:38**

Evan Lenz
1,605 2 8

Great Answer! Thanks for all of the variations and clear explanations. Hopefully this answer will help lots of people in the future! (this could have been broken into several answers) – Greg Mattes May 11 '09 at 12:06

I'm glad you found it helpful. Thanks for the vote. I'm still learning how to use this website. Should I have provided separate responses? – Evan Lenz May 13 '09 at 7:51

Providing separate answers or one answer with several variations is a matter of taste. Independent answers allow independent voting. For example, maybe I would have accepted an answer that uses apply-templates as the best response, but the community might have preferred an answer using for-each. Other alternatives could have been down-voted. My accepted answer would be shown first and the community answer second when sorting by votes. Comments could be addressed to particular solutions. – Greg Mattes May 13 '09 at 19:42

Makes perfect sense. Thanks for the tips! – Evan Lenz May 15 '09 at 6:04

feedback

One such XPath 1.0 expression is:

```
/*/vertex[not(@name = /*/vertex/directed-edge-to/@vertex)]
```

Then just put it into an XSLT stylesheet like that:

```
<xsl:stylesheet version="1.0"
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
 <xsl:output omit-xml-declaration="yes" indent="yes"/>

    <xsl:template match="/">
      <minimal-vertices>
          <xsl:for-each select=
          "/*/vertex[not(@name = /*/vertex/directed-edge-to/@vertex)]"
          >
           <minimal-vertex name="{@name}"/>
          </xsl:for-each>
      </minimal-vertices>
    </xsl:template>
</xsl:stylesheet>
```

When this stylesheet is applied on the originally-provided XML document:

```
<dag>
    <vertex name="A">
        <directed-edge-to vertex="C"/>
    </vertex>
    <vertex name="B">
        <directed-edge-to vertex="C"/>
        <directed-edge-to vertex="D"/>
    </vertex>
    <vertex name="C">
        <directed-edge-to vertex="E"/>
    </vertex>
    <vertex name="D">
        <directed-edge-to vertex="E"/>
    </vertex>
    <vertex name="E">
        <directed-edge-to vertex="G"/>
    </vertex>
    <vertex name="F">
        <directed-edge-to vertex="G"/>
    </vertex>
    <vertex name="G"/>
</dag>
```

The wanted result is produced:

```
<minimal-vertices>
  <minimal-vertex name="A" />
  <minimal-vertex name="B" />
  <minimal-vertex name="F" />
</minimal-vertices>
```

Do note: *A solution for traversing full (maybe cyclic) graphs is available in XSLT* **here**.

answered **May 10 '09 at 13:51**

Dimitre Novatchev
71.5k 5 26 59

Thanks! This is a great answer too, it's very focused on the question that I asked. It was a tough decision, but I accepted Evan's answer because of the breadth of his answer. I am curious about why you prefer the /*/ syntax to //, is there any advantage with the extra character? – Greg Mattes May 11 '09 at 11:57

@greg-mattes THe "//" abbreviation should be avoided whenever possible as it is very expensive, causing the whole subtree rooted at the context node to be searched. "//" at the top level causes the whole XML document to be searched. It is very important *not* to use "//" whenever the structure of the XML document is known at the time of writing the XPath expression. – Dimitre Novatchev May 11 '09 at 16:43

So /*/ is better in general because it limits the search to a single level because * means "selects all element children of the context node" (w3.org/TR/xpath#path-abbrev) rather than all descendants which could be a large search? In this particular example it shouldn't make a difference, but it's a good point to keep in mind. Thanks again. – Greg Mattes May 11 '09 at 19:17

1 I agree with Dimitre about the use of "//". And you're right that performance isn't much of a consideration for this particular data. However, another reason to use /*/vertex, or, even better, /dag/vertex, is that it makes your intentions more explicit. "*" implies that the document element's name may vary, and "//" implies that <vertex> elements might appear as deeper descendants. You can save people reading your code from having to wonder such things by making your intentions more explicit. "//" is still useful, of course, when it's necessary, i.e. when it actually is your intention. – Evan Lenz May 13 '09 at 7:57