NAME

tcldot – graph manipulation in tcl

SYNOPSIS

#!/usr/local/bin/tclsh
package require **Tcldot**

USAGE

Requires the dynamic loading facilities of tcl7.6 or later.

INTRODUCTION

tcldot is a tcl dynamically loaded extension that incorporates the directed graph facilities of **dot(1),** and the undirected graph facilities of **neato(1),** into tcl and provides a set of commands to control those facilities. **tcldot** converts **dot** and **neato** from batch processing tools to an interpreted and, if needed, interactive set of graph manipulation facilities.

COMMANDS

tcldot initially adds only three commands to tcl, namely **dotnew, dotread,** and **dotstring.** These commands return a handle for the graph that has just been created and that handle can then be used as a command for further actions on the graph.

All other "commands" are of the form:

handle **<method>** *parameters*

Many of the methods return further handles of graphs, nodes of edges, which are themselves registered as commands.

The methods are described in detail below, but in summary:

Graph methods are:

addedge, addnode, addsubgraph, countedges, countnodes, layout, listattributes, listedgeattributes, listnodeattributes, listedges, listnodes, listnodesrev, listsubgraphs, render, rendergd, queryattributes, queryedgeattributes, querynodeattributes, queryattributevalues, queryedgeattributevalues, querynodeattributevalues, setattributes, setedgeattributes, setnodeattributes, showname, write.

Node methods are:

addedge, listattributes, listedges, listinedges, listoutedges, queryattributes, queryattributevalues, setattributes, showname.

Edge methods are:

delete, listattributes, listnodes, queryattributes, queryattributevalues, setattributes, showname.

dotnew *graphType ?attributeName attributeValue? ?...?*

creates a new empty graph and returns its *graphHandle.*

graphType can be any supported by **dot(1)** namely: "graph," "digraph," "graphstrict," or "digraphstrict." (In digraphs edges have a direction from tail to head. "Strict" graphs or digraphs collapse multiple edges between the same pair of nodes into a single edge.)

Following the mandatory *graphType* parameter the **dotnew** command will accept an arbitrary

number of attribute name/value pairs for the graph. Certain special graph attributes and permitted values are described in **dot(1),** but the programmer can arbitrarily invent and assign values to additional attributes beyond these. In **dot** the attribute name is separated from the value by an "=" character. In **tcldot** the "=" has been replaced by a " " (space) to be more consistent with **tcl** syntax. e.g.

```
set g [dotnew digraph rankdir LR]
```

dotread *fileHandle*

reads in a dot-language description of a graph from a previously opened file identified by the *fileHandle.* The command returns the *graphHandle* of the newly read graph. e.g.

```
set f [open test.dot r]
set g [dotread $f]
```

dotstring *string*

reads in a dot-language description of a graph from a Tcl string; The command returns the *graphHandle* of the newly read graph. e.g.

```
set g [dotstring $dotsyntaxstring]
```

graphHandle **addnode** *?nodeName? ?attributeName attributeValue? ?...?*

creates a new node in the graph whose handle is *graphHandle* and returns its *nodeHandle.* The handle of a node is a string like: "node0" where the integer value is different for each node. There can be an arbitrary number of attribute name/value pairs for the node. Certain special node attributes and permitted values are described in **dot(1),** but the programmer can arbitrarily invent and assign values to additional attributes beyond these. e.g.

```
set n [$g addnode "N" label "Top\nNode" shape triangle eggs easyover]
```

A possible cause of confusion in **tcldot** is the distinction between handles, names, labels, and variables. The distinction is primarily in who owns them. Handles are owned by tcldot and are guaranteed to be unique within one interpreter session. Typically handles are assigned to variables, like "n" above, for manipulation within a tcl script. Variables are owned by the programmer. Names are owned by the application that is using the graph, typically names are important when reading in a graph from an external program or file. Labels are the text that is displayed with the node (or edge) when the graph is displayed, labels are meaningful to the reader of the graph. Only the handles and variables are essential to **tcldot's** ability to manipulate abstract graphs. If a name is not specified then it defaults to the string representation of the handle, if a label is not specified then it defaults to the name.

graphHandle **addedge** *tailNode headNode ?attributeName attributeValue? ?...?*

creates a new edge in the graph whose handle is *graphHandle* and returns its **edgeHandle.** *tailNode* and *headNode* can be specified either by their *nodeHandle* or by their *nodeName.* e.g.

```
set n [$g addnode]
set m [$g addnode]
$g addedge $n $m label "NM"
```

```
$g addnode N
$g addnode M
$g addedge N M label "NM"
```

The argument is recognized as a handle if possible and so it is best to avoid names like "node6" for nodes. If there is potential for conflict then use **findnode** to translate explicitly from names to handles. e.g.

```
$g addnode "node6"
$g addnode "node99"
$g addedge [$g findnode "node6"] [$g findnode "node99"]
```

There can be an arbitrary number of attribute name/value pairs for the edge. Certain special edge attributes and permitted values are described in **dot(1),** but the programmer can arbitrarily invent and assign values to additional attributes beyond these.

graphHandle **addsubgraph** *?graphName? ?attributeName attributeValue? ?...?*

creates a new subgraph in the graph and returns its *graphHandle.* If the *graphName* is omitted then the name of the subgraph defaults to it's *graphHandle.* There can be an arbitrary number of attribute name/value pairs for the subgraph. Certain special graph attributes and permitted values are described in **dot(1),** but the programmer can arbitrarily invent and assign values to additional attributes beyond these. e.g.

```
set sg [$g addsubgraph dinglefactor 6]
```

Clusters, as described in **dot(1),** are created by giving the subgraph a name that begins with the string: "cluster". Cluster can be labelled by using the *label* attibute. e.g.

```
set cg [$g addsubgraph cluster_A label dongle dinglefactor 6]
```

nodeHandle **addedge** *headNode ?attributeName attributeValue? ?...?*

creates a new edge from the tail node identified by tha *nodeHandle* to the *headNode* which can be specified either by *nodeHandle* or by *nodeName* (with preference to recognizing the argument as a handle). The graph in which this is drawn is the graph in which both nodes are members. There can be an arbitrary number of attribute name/value pairs for the edge. These edge attributes and permitted values are described in **dot(1).** e.g.

```
[$g addnode] addedge [$g addnode] label "NM"
```

graphHandle **delete**

nodeHandle **delete**

edgeHandle **delete**

Delete all data structures associated with the graph, node or edge from the internal storage of the interpreter. Deletion of a node also results in the the deletion of all subtending edges on that node. Deletion of a graph also results in the deletion of all nodes and subgraphs within that graph (and hence all edges too). The return from these delete commands is a null string.

graphHandle **countnodes**

graphHandle **countedges**

Returns the number of nodes, or edges, in the graph.

graphHandle **listedges**

graphHandle **listnodes**

graphHandle **listnodesrev**

graphHandle **listsubgraphs**

nodeHandle **listedges**

nodeHandle **listinedges**

nodeHandle **listoutedges**

edgeHandle **listnodes**

Each return a list of handles of graphs, nodes or edges, as appropriate.

graphHandle **findnode** *nodeName*

graphHandle **findedge** *tailnodeName headNodeName*

nodeHandle **findedge** *nodeName*

Each return the handle of the item if found, or an error if none are found. For non-strict graphs when there are multiple edges between two nodes **findedge** will return an arbitrary edge from the set.

graphHandle **showname**

nodeHandle **showname**

edgeHandle **showname**

Each return the name of the item. Edge names are of the form: "a−>b" where "a" and "b" are the names of the nodes and the connector "−>" indicates the tail-to-head direction of the edge. In undirected graphs the connector "−−" is used.

graphHandle **setnodeattributes** *attributeName attributeValue ?...?*

graphHandle **setedgeattributes** *attributeName attributeValue ?...?*

Set one or more default attribute name/values that are to apply to all nodes (edges) unless overridden by subgraphs or per-node (per-edge) attributes.

graphHandle **listnodeattributes**

graphHandle **listedgeattributes**

Return a list of attribute names.

graphHandle **querynodeattributes** *attributeName ?...?*

graphHandle **queryedgeattributes** *attributeName ?...?*

Return a list of default attribute value, one value for each of the attribute names provided with the command.

graphHandle **querynodeattributes** *attributeName ?...?*

graphHandle **queryedgeattributes** *attributeName ?...?*

Return a list of pairs of attrinute name and default attribute value, one pair for each of the attribute names provided with the command.

graphHandle **setattributes** *attributeName attributeValue ?...?*

nodeHandle **setattributes** *attributeName attributeValue ?...?*

edgeHandle **setattributes** *attributeName attributeValue ?...?*

Set one or more attribute name/value pairs for a specific graph, node, or edge instance.

graphHandle **listattributes**

nodeHandle **listattributes**

edgeHandle **listattributes**

Return a list of attribute names (attribute values are provided by **queryattribute**

graphHandle **queryattributes** *attributeName ?...?*

nodeHandle **queryattributes** *attributeName ?...?*

edgeHandle **queryattributes** *attributeName ?...?*

Return a list of attribute value, one value for each of the attribute names provided with the command.

graphHandle **queryattributevalues** *attributeName ?...?*

nodeHandle **queryattributevalues** *attributeName ?...?*

edgeHandle **queryattributevalues** *attributeName ?...?*

Return a list of pairs or attribute name and attribute value, one value for each of the attribute names provided with the command.

graphHandle **layout ?dot|neato|circo|twopi|fdp|nop?**

Annotate the graph with layout information. This commands takes an abstract graph add shape and position information to it according to the layout engine's rules of eye-pleasing graph layout. If the layout engine is unspecified then it defaults to **dot** for directed graphs, and **neato** otherwise. If the **nop** engine is specified then layout information from the input graph is used. The result of the layout is stored as additional attributes name/value pairs in the graph, node and edges. These attributes are intended to be interpreted by subsequent *write* or *render* commands.

graphHandle **write** *fileHandle format ?dot|neato|circo|twopi|fdp|nop?*

Write a graph to the open file represented by *fileHandle* in a specific *format*. Possible *formats* are: "ps" "mif" "plain" "dot" "gif" "ismap" If the layout hasn't been already done, then it will be done as part of this operation using the same rules for selecting the layout engine as for the layout command.

graphHandle **rendergd** *gdHandle*

Generates a rendering of a graph to a new or existing gifImage structure (see **gdTcl(1)**). Returns the *gdHandle* of the image. If the layout hasn't been already done, then it will be done as part of this operation using the same rules for selecting the layout engine as for the layout command.

graphHandle **render** *?canvas ?dot|neato|circo|twopi|fdp|nop??*

If no *canvas* argument is provided then **render** returns a string of commands which, when evaluated, will render the graph to a **Tk** canvas whose *canvasHandle* is available in variable **$c**

If a *canvas* argument is provided then **render** produces a set of commands for *canvas* instead of $c.

If the layout hasn't been already done, then it will be done as part of this operation using the same rules for selecting the layout engine as for the layout command.

```
#!/usr/local/bin/wish
package require Tcldot
set c [canvas .c]
pack $c
set g [dotnew digraph rankdir LR]
$g setnodeattribute style filled color white
[$g addnode Hello] addedge [$g addnode World!]
$g layout
if {[info exists debug]} {
   puts [$g render]        ;# see what render produces
}
eval [$g render]
```

Render generates a series of canvas commands for each graph element, for example a node typically consist of two items on the canvas, one for the shape and the other for the label. The canvas items are automatically *tagged* (See **canvas(n)**) by the commands generated by render. The tags take one of two forms: text items are tagged with 0<handle> and shapes and lines are rendered with 1<handle>.

The tagging can be used to recognize when a user wants to interact with a graph element using the mouse. See the script in *examples/disp* of the tcldot distribution for a demonstration of this facility.

BUGS

Still batch-oriented. It would be nice if the layout was maintained incrementally. (The intent is to address this limitation in graphviz_2_0.)

AUTHOR

John Ellson (ellson@graphviz.org)

ACKNOWLEDGEMENTS

John Ousterhout, of course, for **tcl** and **tk.** Steven North and Eleftherios Koutsofios for **dot.** Karl Lehenbauer and Mark Diekhans of NeoSoft for the handles.c code which was derived from tclXhandles.c. Tom Boutell of the Quest Center at Cold Spring Harbor Labs for the gif drawing routines. Spencer Thomas of the University of Michigan for gdTcl.c. Dayatra Shands for coding much of the initial implementation of **tcldot.**

KEYWORDS

graph, tcl, tk, dot, neato.