New Stuff for VIVA v2.2

Functional changes

Comparison operators and functions now also work on strings.

New numeric/comparison functions

Signature	Returns	Description
`abs(numeric)`	Parameter Type	Absolute Value
`max(arg,arg,…)` can take any number of parameters of any type, but they must all be of the same type	Parameter Type	Maximum
`maxall(arg,arg,…)` can take any number of parameters of any type, but they must all be of the same type	Parameter Type	Maximum (Vector function)
`min(arg,arg,…)` can take any number of parameters of any type, but they must all be of the same type	Parameter Type	Minimum
`minall(arg,arg,…)` can take any number of parameters of any type, but they must all be of the same type	Parameter Type	Minimum (Vector function)

New Global Value functions

Signature	Returns	Description
addtolist(id,value) **id** and **value** can be any type	boolean	Add the **value** to the **id** list (It's actually a set) Always returns `true`
getdouble(id) **id** can be any type	double	Retrieve a stored double by **id**
getfloat(id) **id** can be any type	float	Retrieve a stored float by **id**
getint(id) **id** can be any type	int	Retrieve a stored integer by **id**
getlong(id) **id** can be any type	long	Retrieve a stored long by **id**
getstring(id) **id** can be any type	string	Retrieve a stored double by **id**
inlist(id,value) **id** and **value** can be any type	boolean	Determines if the **value** is in the **id** list (It's actually a set)
setdouble(id,double) **id** can be any type	boolean	Store a double by **id** Always returns **true**
setfloat(id,float) **id** can be any type	boolean	Store a float by **id** Always returns **true**
setint(id,integer) **id** can be any type	boolean	Store an integer by **id** Always returns **true**
setlong(id,long) **id** can be any type	boolean	Store a long by **id** Always returns **true**
setstring(id,string) **id** can be any type	boolean	Store a string by **id** Always returns **true**

Graphs!

<table>
<tr><th>Signature</th><th>Returns</th><th>Description</th></tr>
<tr><td>graph(graph_id(,parameter)*)

graph_id must be a string

parameters must be specific strings</td><td>boolean</td><td>Create a graph with the given graph_id

Parameters:

<table>
<tr><td>“directed”</td><td>Directed graph</td></tr>
<tr><td>“undirected”</td><td>*Undirected graph</td></tr>
<tr><td>“weighted”</td><td>Weighted edges</td></tr>
<tr><td>“unweighted”</td><td>*Unweighted edges</td></tr>
<tr><td>“multi”</td><td>Allow duplicate edges</td></tr>
<tr><td>“nomulti”</td><td>*No duplicate edges</td></tr>
<tr><td>“self”</td><td>Allow self edges</td></tr>
<tr><td>“noself”</td><td>*No self edges</td></tr>
<tr><td>“auto”</td><td>addedge() can add nodes</td></tr>
<tr><td>“noauto”</td><td>*addedge() fails if node not already added</td></tr>
</table>
*These options are the defaults, and don’t need to be specified. They are included only for completeness, and in case they are dynamically specified in the input.

Fails if contradictory parameters are given

Otherwise, always returns true</td></tr>
<tr><td>addnode(graph_id,node_id)

graph_id must be a string

node_id must be discrete (int, long or string)</td><td>boolean</td><td>Add a node to the given graph

The set of nodes is, indeed, a set, so adding a duplicate node is a no-op

Always returns true</td></tr>
<tr><td>addnodes(graph_id,start,end)

graph_id must be a string

start and end must be integers</td><td>boolean</td><td>Add a list of nodes to the given graph, with node_ids [start..end] inclusive

The set of nodes is, indeed, a set, so adding a duplicate node is a no-op

Always returns true</td></tr>
<tr><td>addedge(graph_id,from,to[,weight])

graph_id must be a string

from and to must be discrete (int, long or string)

weight must be numeric</td><td>boolean</td><td>Add an edge to the given graph.

Fails adding a weight to an unweighted graph

Fails not adding a weight in a weighted graph

If no errors, always returns true</td></tr>
<tr><td>components(graph_id)</td><td>integer</td><td>The number of connected components in the</td></tr>
</table>

graph_id must be a string		graph
iscactus(graph_id) **graph_id** must be a string	boolean	**true** if the graph is a Cactus
isconnected(graph_id) **graph_id** must be a string	boolean	**true** if the graph is a single Connected Component
isdag(graph_id) **graph_id** must be a string	boolean	**true** if the graph is a Directed Acyclic Graph
isdesert(graph_id) **graph_id** must be a string	boolean	**true** if the graph is a Desert (every Connected Component is a Cactus)
isforest(graph_id) **graph_id** must be a string	boolean	**true** if the graph is a Forest (every Connected Component is a Tree)
istree(graph_id) **graph_id** must be a string	boolean	**true** if the graph is a Tree
nonegcycles(graph_id) **graph_id** must be a string	boolean	Test if the graph has no negative cycles Always **true** if the graph is unweighted (even if the graph is undirected) Fails if the graph is weighted and undirected Fails if $\|V\| \cdot \|E\| > 1{,}000{,}000$ Otherwise, returns **true** if the graph has no negative cycles Testing for no negative cycles in an undirected graph is NP-Complete. The fastest algorithm for testing for negative cycles uses Bellman/Ford, which is $O(\|V\| \cdot \|E\|)$

Note: Under some conditions, some of the Graph functions “Fail”. When this happens, they do not simply return **false**. They print a message to the output stream, and VIVA stops processing for that input file.

Here is a sample input statements from the recent 2020 NAC:

The first line contains two space-separated integers n and q ($1 \leq n, q \leq 2 \cdot 10^5$), where n is the number of nodes in the tree and q is the number of queries to be answered. The nodes are numbered from 1 to n.

Each of the next $n-1$ lines contains two space-separated integers u and v ($1 \leq u, v \leq n, u \neq v$), indicating an undirected edge between nodes u and v. It is guaranteed that this set of edges forms a valid tree.

Each of the next q lines contains two space-separated integers r and p ($1 \leq r, p \leq n$), which are the nodes of the roots for the given query.

Here is a possible VIVA pattern for this input:

```
n q (1<=n<=200000, 1<=q<=200000, graph("x"), addnodes("x",1,n));
{[*n-1]
      u v (1<=u<=n, 1<=v<=n, u!=v, addedge("x",u,v));
[istree("x")]}
{[*q]
      r p (1<=r<=n,1<=p<=n);
}
```

Note that is isn't necessary to specify the constraints on u and v, as **`addedge()`** will fail if any of those constraints are violated. But, if **`addedge()`** fails, VIVA stops processing that file, so this may be more graceful. Then again, if **`addedge()`** fails, its message is much more informative.