

The `lt3graph` package^{*†}

Michiel Helvensteijn
mhelvens+latex@gmail.com

April 22, 2014

Development of this package is organized at github.com/mhelvens/latex-lt3graph.
I am happy to receive feedback there!

1 Introduction

This package provides a data-structure for use in the $\text{\LaTeX}3$ programming environment. It allows you to represent a *directed graph*, which contains *vertices* (nodes), and *edges* (arrows) to connect them.¹ One such a graph is defined below:

```
\ExplSyntaxOn
\graph_new:N          \l_my_graph
\graph_put_vertex:Nn  \l_my_graph {v}
\graph_put_vertex:Nn  \l_my_graph {w}
\graph_put_vertex:Nn  \l_my_graph {x}
\graph_put_vertex:Nn  \l_my_graph {y}
\graph_put_vertex:Nn  \l_my_graph {z}
\graph_put_edge:Nnn   \l_my_graph {v} {w}
\graph_put_edge:Nnn   \l_my_graph {w} {x}
\graph_put_edge:Nnn   \l_my_graph {w} {y}
\graph_put_edge:Nnn   \l_my_graph {w} {z}
\graph_put_edge:Nnn   \l_my_graph {y} {z}
\graph_put_edge:Nnn   \l_my_graph {z} {x}
\ExplSyntaxOff
```

Each vertex is identified by a *key*, which, to this library, is a string: a list of characters with category code 12 and spaces with category code 10. An edge is then declared between two vertices by referring to their keys.

We could then, for example, use TikZ to draw this graph:

^{*}This document corresponds to `lt3graph` v0.1.3, dated 2014/04/21.

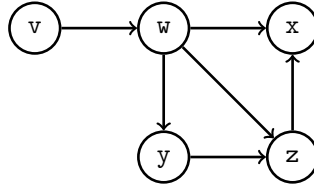
[†]The prefix `lt3` indicates that this package is a user-contributed `expl3` library, in contrast to packages prefixed with `l3`, which are officially supported by the $\text{\LaTeX}3$ team.

¹ Mathematically speaking, a directed graph is a tuple (V, E) with a set of vertices V and a set of edges $E \subseteq V \times V$ connecting those vertices.

```

\centering
\begin{tikzpicture}[every path/.style={line width=1pt,->}]
  \newcommand{\vrt}[1]{\node[#1]{\ttfamily\vphantom{Iy}#1}; }
  \matrix[nodes={circle,draw},
    row sep=1cm, column sep=1cm,
    execute at begin cell=\vrt] {
    v & w & x \\
    & y & z \\
  };
  \ExplSyntaxOn
  \graph_map_edges_inline:Nn \l_my_graph
  { \draw (#1) to (#2); }
  \ExplSyntaxOff
\end{tikzpicture}

```



Just to be clear, this library is *not about drawing* graphs. It does not, inherently, understand any TikZ. It is about *representing* graphs. This allows us do perform analysis on their structure. We could, for example, determine if there is a cycle in the graph:

```

\ExplSyntaxOn
  \graph_if_cyclic:NTF \l_my_graph {Yep} {Nope}
\ExplSyntaxOff

```

Nope

Indeed, there are no cycles in this graph. We can also list its vertices in topological order:

```

\ExplSyntaxOn
  \clist_new:N \LinearClist
  \graph_map_topological_order_inline:Nn \l_my_graph
  { \clist_put_right:Nn \LinearClist {\texttt{#1}} }
\ExplSyntaxOff

```

Visiting dependencies first: \(\ \LinearClist \)

Visiting dependencies first: v,w,y,z,x

There is a great deal more that can be done with graphs (some of which is even implemented in this library). A common use-case will be to attach data to vertices and/or edges. You could accomplish this with a property map from `l3prop`, but this library has already done that for you! Every vertex and every edge can store arbitrary token lists.²

In the next example we store the *degree* (the number of edges, both incoming and outgoing) of each vertex inside that vertex as data. We then query all vertices directly reachable from `w` and print their information in the output stream:

²This makes the mathematical representation of our graphs actually a 4-tuple (V, E, v, e) , where $v : V \rightarrow TL$ is a function that maps every vertex to a token list and $e : E \rightarrow TL$ is a function that maps every edge (i.e., pair of vertices) to a token list.

```

\ExplSyntaxOn
  \cs_generate_variant:Nn \graph_put_vertex:Nnn {Nnf}
  \graph_map_vertices_inline:Nn \l_my_graph {
    \graph_put_vertex:Nnf \l_my_graph {#1}
    { \graph_get_degree:Nn \l_my_graph {#1} }
  }
\ExplSyntaxOff

```

It's just an additional parameter on the `\graph_put_vertex` function. Edges can store data in the same way:

```

\ExplSyntaxOn
  \graph_map_edges_inline:Nn \l_my_graph {
    \graph_put_edge:Nnnn \l_my_graph {#1} {#2}
    { \int_eval:n{##1 * ##2} }
  }
\ExplSyntaxOff

```

The values `##1` and `##2` represent the data stored in, respectively, vertices `#1` and `#2`. This is a feature of `\graph_put_edge:Nnnn` added for your convenience.

We can show the resulting graph in a table, which is handy for debugging:

```

\ExplSyntaxOn \centering
  \graph_display_table:N \l_my_graph
\ExplSyntaxOff

```

		v	w	x	y	z
v	1		4	(tr)	(tr)	(tr)
w	4			8	8	12
x	2					
y	2			(tr)		6
z	3			6		

The green cells represent edges directly connecting two vertices. The yellow cells don't have edges, but indicate that there is a sequence of edges connecting two vertices transitively.

Two vertices can have at most two arrows connecting them: one for each direction. If you want to represent a *multidigraph* (or *quiver*; I'm not making this up), you could consider storing a (pointer to a) list at each edge.

Finally, we demonstrate some transformation functions. The first generates the transitive closure of a graph:

```

\ExplSyntaxOn
  \graph_new:N \l_closed_graph
  \cs_new:Nn \__closure_combiner:nnn { #1,~#2,~(#3) }
  \graph_set_transitive_closure:NNNn
    \l_closed_graph \l_my_graph
    \__closure_combiner:nnn {--}
\ExplSyntaxOff

```

```

\ExplSyntaxOn \centering
\graph_display_table:N \l_my_graph
\(\ \ \rightsquigarrow\ \ \)
\graph_display_table:Nn \l_closed_graph
{ row_keys = false, vertex_vals = false }
\ExplSyntaxOff

```

		v	w	x	y	z
v	1		4	(tr)	(tr)	(tr)
w	4			8	8	12
x	2					
y	2			(tr)		6
z	3			6		

 \rightsquigarrow

v	w	x	y	z
	4	4, 8, (-)	4, 8, (-)	4, 12, (-)
		12, 6, (8)	8	8, 6, (12)
		6, 6, (-)		6
		6		

There is a simpler version (`\graph_set_transitive_closure:NN`) that sets the values of the new edges to the empty token-list. The demonstrated version takes an expandable function to determine the new value, which has access to the values of the two edges being combined (as #1 and #2), as well as the value of the possibly already existing transitive edge (as #3). If there was no transitive edge there already, the value passed as #3 is the fourth argument of the transformation function; in this case --.

The second transformation function generates the transitive reduction:

```

\ExplSyntaxOn
\graph_new:N \l_reduced_graph
\graph_set_transitive_reduction:NN
\l_reduced_graph \l_my_graph
\ExplSyntaxOff

```

```

\ExplSyntaxOn \centering
\graph_display_table:N \l_my_graph
\(\ \ \rightsquigarrow\ \ \)
\graph_display_table:Nn \l_reduced_graph
{ row_keys = false, vertex_vals = false }
\ExplSyntaxOff

```

		v	w	x	y	z
v	1		4	(tr)	(tr)	(tr)
w	4			8	8	12
x	2					
y	2			(tr)		6
z	3			6		

 \rightsquigarrow

v	w	x	y	z
	4	(tr)	(tr)	(tr)
		(tr)	8	(tr)
		(tr)		6
		6		

2 API Documentation

Sorry! There is no full API documentation yet. But in the meantime, much of the API is integrated in the examples of the previous section, and everything is documented (however sparsely) in the implementation below.

3 Implementation

We now show and explain the entire implementation from `lt3graph.sty`.

3.1 Package Info

```
1 \NeedsTeXFormat{LaTeX2e}
2 \RequirePackage{expl3}
3 \ProvidesExplPackage{lt3graph}{2014/04/21}{0.1.3}
4 {a LaTeX3 datastructure for representing directed graphs with data}
```

3.2 Required Packages

These are the packages we'll need:

```
5 \RequirePackage{l3candidates}
6 \RequirePackage{l3clist}
7 \RequirePackage{l3keys2e}
8 \RequirePackage{l3msg}
9 \RequirePackage{l3prg}
10 \RequirePackage{l3prop}
11 \RequirePackage{xparse}
12 \RequirePackage{withargs}
```

3.3 Additions to L^AT_EX3 Fundamentals

These are three macros for working with ‘set literals’ in an expandable context. They use internal macros from `l3prop...` Something I'm really not supposed to do.

```
13 \prg_new_conditional:Npnn \__graph_set_if_in:nn #1#2 { p }
14 {
15   \__prop_if_in:nwn {#2} #1 \s_obj_end
16   \__prop_pair:wn #2 \s__prop { }
17   \q_recursion_tail
18   \__prg_break_point:
19 }
20
21 \cs_set_eq:NN \__graph_empty_set \s__prop
22
23 \cs_new:Nn \__graph_set_cons:nn {
24   #1 \__prop_pair:wn #2 \s__prop {}
25 }
```

3.4 Data Access

These functions generate the multi-part csnames under which all graph data is stored:

```
26 \cs_new:Nn \__graph_t1:n { g__graph_data (#1) _t1 }
27 \cs_new:Nn \__graph_t1:nn { g__graph_data (#1) (#2) _t1 }
28 \cs_new:Nn \__graph_t1:nnn { g__graph_data (#1) (#2) (#3) _t1 }
29 \cs_new:Nn \__graph_t1:nnnn { g__graph_data (#1) (#2) (#3) (#4) _t1 }
30 \cs_new:Nn \__graph_t1:nnnnn { g__graph_data (#1) (#2) (#3) (#4) (#5) _t1 }
```

The following functions generate multi-part keys to use in property maps:

```

31 \cs_new:Nn \__graph_key:n      { key (#1) }
32 \cs_new:Nn \__graph_key:nn     { key (#1) (#2) }
33 \cs_new:Nn \__graph_key:nnn    { key (#1) (#2) (#3) }
34 \cs_new:Nn \__graph_key:nnnn   { key (#1) (#2) (#3) (#4) }
35 \cs_new:Nn \__graph_key:nnnnn  { key (#1) (#2) (#3) (#4) (#5) }

```

A quick way to iterate through property maps holding graph data:

```

36 \cs_new_protected:Nn \__graph_for_each_prop_datatype:n
37 { \seq_map_inline:Nn \g__graph_prop_data_types_seq {#1} }
38 \seq_new:N          \g__graph_prop_data_types_seq
39 \seq_set_from_clist:Nn \g__graph_prop_data_types_seq
40 {vertices, edge-values, edge-froms, edge-tos,
41  edge-triples, indegree, outdegree}

```

3.5 Storing data through pointers

The following function embodies a L^AT_EX3 design pattern for representing non-null pointers. This allows data to be 'protected' behind a macro redirection. Any number of expandable operations can be applied to the pointer indiscriminately without altering the data, even when using :x, :o or :f expansion. Expansion using :v dereferences the pointer and returns the data exactly as it was passed through #2. Expansion using :c returns a control sequence through which the data can be modified.

```

42 \cs_new_protected:Nn \__graph_ptr_new:Nn {
43   \withargs [\uniquecsname] {
44     \tl_set:Nn #1 {##1}
45     \tl_new:c   {##1}
46     \tl_set:cn  {##1} {#2}
47   }
48 }
49 \cs_new_protected:Nn \__graph_ptr_gnew:Nn {
50   \withargs [\uniquecsname] {
51     \tl_gset:Nn #1 {##1}
52     \tl_new:c   {##1}
53     \tl_gset:cn {##1} {#2}
54   }
55 }

```

3.6 Creating and initializing graphs

Globally create a new graph:

```

56 \cs_new_protected:Nn \graph_new:N {
57   \graph_if_exist:NTF #1 {
58     % TODO: error
59   }{
60     \tl_new:N #1
61     \tl_set:Nf #1 { \tl_trim_spaces:f {\str_tail:n{#1}} }
62     \int_new:c {\__graph_tl:nnn{graph}{#1}{vertex-count}}
63     \__graph_for_each_prop_datatype:n
64     { \prop_new:c {\__graph_tl:nnn{graph}{#1}{##1}} }
65   }

```

```

66 }
67 \cs_generate_variant:Nn \tl_trim_spaces:n {f}

```

Remove all data from a graph:

```

68 \cs_new_protected:Nn \graph_clear:N
69   { \__graph_clear:Nn #1 { } }
70 \cs_new_protected:Nn \graph_gclear:N
71   { \__graph_clear:Nn #1 {g} }
72 \cs_new_protected:Nn \__graph_clear:Nn {
73   \__graph_for_each_prop_datatype:n
74     { \use:c{prop_#2clear:c} { \__graph_tl:nnn{graph}{#1}{##1}} }
75 }

```

Create a new graph if it doesn't already exist, then remove all data from it:

```

76 \cs_new_protected:Nn \graph_clear_new:N
77   { \__graph_clear_new:Nn #1 { } }
78 \cs_new_protected:Nn \graph_gclear_new:N
79   { \__graph_clear_new:Nn #1 {g} }
80 \cs_new_protected:Nn \__graph_clear_new:Nn {
81   \graph_if_exists:NF #1
82   { \graph_new:N #1 }
83   \use:c{graph_#2clear:N} #1
84 }

```

Set all data in graph #1 equal to that in graph #2:

```

85 \cs_new_protected:Nn \graph_set_eq:NN
86   { \__graph_set_eq:NNn #1 #2 { } }
87 \cs_new_protected:Nn \graph_gset_eq:NN
88   { \__graph_set_eq:NNn #1 #2 {g} }
89 \cs_new_protected:Nn \__graph_set_eq:NNn {
90   \use:c{graph_#3clear:N} #1
91   \__graph_for_each_prop_datatype:n
92     {
93       \use:c{prop_#3set_eq:cc}
94       { \__graph_tl:nnn{graph}{#1}{##1}}
95       { \__graph_tl:nnn{graph}{#2}{##1}}
96     }
97 }

```

An expandable test of whether a graph exists. It does not actually test whether the command sequence contains a graph and is essentially the same as `\cs_if_exist:N(TF):`

```

98 \cs_set_eq:NN \graph_if_exist:Np \cs_if_exist:Np
99 \cs_set_eq:NN \graph_if_exist:NT \cs_if_exist:NT
100 \cs_set_eq:NN \graph_if_exist:NF \cs_if_exist:NF
101 \cs_set_eq:NN \graph_if_exist:NTF \cs_if_exist:NTF

```

3.7 Manipulating graphs

Put a new vertex inside a graph:

```

102 \cs_new_protected:Nn \graph_put_vertex:Nn
103 { \__graph_put_vertex:Nnnn #1 {#2} {} { } }
104 \cs_new_protected:Nn \graph_gput_vertex:Nn
105 { \__graph_put_vertex:Nnnn #1 {#2} {} {g} }
106 \cs_new_protected:Nn \graph_put_vertex:Nnn
107 { \__graph_put_vertex:Nnnn #1 {#2} {#3} { } }
108 \cs_new_protected:Nn \graph_gput_vertex:Nnn
109 { \__graph_put_vertex:Nnnn #1 {#2} {#3} {g} }
110 \cs_new_protected:Nn \__graph_put_vertex:Nnnn
111 {
112   %% create pointer to value
113   %
114   \use:c{__graph_ptr_#4new:Nn} \l__graph_vertex_data_tl {#3}
115
116   %% add the vertex
117   %
118   \use:c{prop_#4put:cnV} {\__graph_tl:nnn{graph}{#1}{vertices}}
119   {#2} \l__graph_vertex_data_tl
120
121   %% increment the vertex counter
122   %
123   \use:c{int_#4incr:c} {\__graph_tl:nnn{graph}{#1}{vertex-count}}
124
125   \graph_get_vertex:NnNT #1 {#2} \l_tmpa_tl {
126     %% initialize degree to 0
127     %
128     \use:c{prop_#4put:cn} {\__graph_tl:nnn{graph}{#1}{indegree}} {#2} {0}
129     \use:c{prop_#4put:cn} {\__graph_tl:nnn{graph}{#1}{outdegree}} {#2} {0}
130   }
131 }
132 \tl_new:N \l__graph_vertex_data_tl

```

Put a new edge inside a graph:

```

133 \cs_new_protected:Nn \graph_put_edge:Nnn
134 { \__graph_put_edge:Nnnnn #1 {#2} {#3} {} { } }
135 \cs_new_protected:Nn \graph_gput_edge:Nnn
136 { \__graph_put_edge:Nnnnn #1 {#2} {#3} {} {g} }
137 \cs_new_protected:Nn \graph_put_edge:Nnnn
138 { \__graph_put_edge:Nnnnn #1 {#2} {#3} {#4} { } }
139 \cs_new_protected:Nn \graph_gput_edge:Nnnn
140 { \__graph_put_edge:Nnnnn #1 {#2} {#3} {#4} {g} }
141 \cs_new_protected:Nn \__graph_put_edge:Nnnnn
142 {
143   \graph_get_vertex:NnNTF #1 {#2} \l__graph_from_value_tl {
144     \graph_get_vertex:NnNTF #1 {#3} \l__graph_to_value_tl {
145       \graph_get_edge:NnnNF #1 {#2} {#3} \l_tmpa_tl {
146         %% increment outgoing degree of vertex #2
147         %
148         \use:c{prop_#5put:cnf} {\__graph_tl:nnn{graph}{#1}{outdegree}} {#2}
149         {\int_eval:n {
150           \prop_get:cn {\__graph_tl:nnn{graph}{#1}{outdegree}} {#2} + 1
151         }}
152

```



```

153         %% increment incoming degree of vertex #3
154         %
155         \use:c{prop_#5put:cnf} {\__graph_tl:nnn{graph}{#1}{indegree}} {#3}
156         {\int_eval:n {
157             \prop_get:cn {\__graph_tl:nnn{graph}{#1}{indegree}} {#3} + 1
158         }}
159     }
160
161     %%% actually add the edge
162     %
163     \withargs:VVn \l__graph_from_value_tl \l__graph_to_value_tl {
164         \use:c{prop_#5put:cox}
165         { \__graph_tl:nnn{graph}{#1}{edge-froms} }
166         { \__graph_key:nn{#2}{#3} }
167         { \tl_to_str:n{#2} }
168         \use:c{prop_#5put:cox}
169         { \__graph_tl:nnn{graph}{#1}{edge-tos} }
170         { \__graph_key:nn{#2}{#3} }
171         { \tl_to_str:n{#3} }
172         \use:c{\__graph_ptr_#5new:Nn} \l__graph_edge_data_tl {#4}
173         \use:c{prop_#5put:coV}
174         { \__graph_tl:nnn{graph}{#1}{edge-values} }
175         { \__graph_key:nn{#2}{#3} }
176         \l__graph_edge_data_tl
177         \use:c{prop_#5put:cox}
178         { \__graph_tl:nnn{graph}{#1}{edge-triples} }
179         { \__graph_key:nn{#2}{#3} }
180         { {\tl_to_str:n{#2}} }
181         { {\tl_to_str:n{#3}} }
182         { \l__graph_edge_data_tl }
183     }
184     }{
185         % TODO: Error ('to' vertex doesn't exist)
186     }
187     }{
188         % TODO: Error ('from' vertex doesn't exist)
189     }
190 }
191 \cs_generate_variant:Nn \prop_gput:Nnn {cox, coV, cnf}
192 \cs_generate_variant:Nn \prop_put:Nnn {cox, coV, cnf}
193 \cs_generate_variant:Nn \withargs:nnn {VVn}
194 \tl_new:N \l__graph_edge_data_tl
195 \tl_new:N \l__graph_from_value_tl
196 \tl_new:N \l__graph_to_value_tl

```

Remove a vertex from a graph, automatically removing any connected edges:

```

197 \cs_new_protected:Nn \graph_remove_vertex:Nn
198 { \__graph_remove_vertex:Nnn #1 {#2} { } }
199 \cs_new_protected:Nn \graph_gremove_vertex:Nn
200 { \__graph_remove_vertex:Nnn #1 {#2} {g} }
201 \cs_new_protected:Nn \__graph_remove_vertex:Nnn
202 {
203     \graph_get_vertex:NnNT #1 {#2} \l__graph_vertex_data_tl {

```

```

204      %%% remove outgoing edges
205      %
206      \graph_map_outgoing_edges_inline:Nnn #1 {#2}
207      { \use:c{graph_#3remove_edge:Nnn} #1 {##1} {##2} }
208
209      %%% remove incoming edges
210      %
211      \graph_map_incoming_edges_inline:Nnn #1 {#2}
212      { \use:c{graph_#3remove_edge:Nnn} #1 {##1} {##2} }
213
214      %%% remove the vertex
215      %
216      \use:c{prop_#3remove:cn} {\_graph_tl:nnn{graph}{#1}{vertices}} {#2}
217      \use:c{prop_#3remove:cn} {\_graph_tl:nnn{graph}{#1}{indegree}} {#2}
218      \use:c{prop_#3remove:cn} {\_graph_tl:nnn{graph}{#1}{outdegree}} {#2}
219
220      %%% decrement the vertex counter
221      %
222      \use:c{int_#3decr:c} {\_graph_tl:nnn{graph}{#1}{vertex-count}}
223    }
224  }
225  \cs_generate_variant:Nn \prop_put:Nnn {cnV}
226  % \tl_new:N \l__graph_vertex_data_tl % reusing from other function

```

Remove an edge from the graph:

```

227  \cs_new_protected:Nn \graph_remove_edge:Nnn
228  { \_graph_remove_edge:Nnnn #1 {#2} {#3} { } }
229  \cs_new_protected:Nn \graph_gremove_edge:Nnn
230  { \_graph_remove_edge:Nnnn #1 {#2} {#3} {g} }
231  \cs_new_protected:Nn \_graph_remove_edge:Nnnn {
232    \graph_get_edge:NnnNT #1 {#2} {#3} \l__graph_edge_data_tl {
233      %%% decrement outdegree of vertex #2
234      %
235      \use:c{prop_#4put:cnf} {\_graph_tl:nnn{graph}{#1}{outdegree}} {#2}
236      { \int_eval:n {
237        \prop_get:cn {\_graph_tl:nnn{graph}{#1}{outdegree}} {#2} - 1
238      } }
239
240      %%% decrement indegree of vertex #3
241      %
242      \use:c{prop_#4put:cnf} {\_graph_tl:nnn{graph}{#1}{indegree}} {#3}
243      { \int_eval:n {
244        \prop_get:cn {\_graph_tl:nnn{graph}{#1}{indegree}} {#3} - 1
245      } }
246
247      %%% actually remove edge
248      %
249      \use:c{prop_#4remove:co}
250      { \_graph_tl:nnn{graph}{#1}{edge-froms} }
251      { \_graph_key:nn{#2}{#3} }
252      \use:c{prop_#4remove:co}
253      { \_graph_tl:nnn{graph}{#1}{edge-to} }
254      { \_graph_key:nn{#2}{#3} }

```

```

255 \use:c{prop_#4remove:co}
256 { \_graph_tl:nnn{graph}{#1}{edge-values} }
257 { \_graph_key:nn{#2}{#3} }
258 \use:c{prop_#4remove:co}
259 { \_graph_tl:nnn{graph}{#1}{edge-triples} }
260 { \_graph_key:nn{#2}{#3} }
261 }
262 }
263 \cs_generate_variant:Nn \prop_remove:Nn {co}
264 \cs_generate_variant:Nn \prop_gremove:Nn {co}
265 \cs_generate_variant:Nn \prop_put:Nnn {cnf}
266 \cs_generate_variant:Nn \prop_gput:Nnn {cnf}
267 %\tl_new:N \l__graph_edge_data_tl % reusing from other function

```

Add all edges from graph #2 to graph #1, but only between nodes already present in #1:

```

268 \cs_new_protected:Nn \graph_put_edges_from:NN
269 { \_graph_gput_edges_from:NNn #1 #2 { } }
270 \cs_new_protected:Nn \graph_gput_edges_from:NN
271 { \_graph_gput_edges_from:NNn #1 #2 {g} }
272 \cs_new_protected:Nn \_graph_gput_edges_from:NNn
273 {
274   \graph_map_edges_inline:Nn #2 {
275     \graph_if_vertex_exist:NnT #1 {##1} {
276       \graph_if_vertex_exist:NnT #1 {##2} {
277         \use:c{graph_#3put_edge:Nnnn} #1 {##1} {##2} {##3}
278       }
279     }
280   }
281 }

```

3.8 Recovering values from graphs with branching

Test whether a vertex #2 exists. If so, its value is stored in #3 and T is left in the input stream. If it doesn't, F is left in the input stream.

```

282 \prg_new_protected_conditional:Nnn \graph_get_vertex:NnN
283 {T, F, TF}
284 {
285   \prop_get:cnNTF { \_graph_tl:nnn {graph} {#1} {vertices} } {#2} #3
286   { \tl_set:Nv #3 {#3} \prg_return_true: }
287   { \prg_return_false: }
288 }

```

Test whether an edge #2-#3 exists. If so, its value is stored in #4 and T is left in the input stream. If it doesn't, F is left in the input stream.

```

289 \prg_new_protected_conditional:Nnn \graph_get_edge:NnnN
290 {T, F, TF}
291 {
292   \prop_get:cnNTF
293   { \_graph_tl:nnn{graph}{#1}{edge-values} }
294   { \_graph_key:nn{#2}{#3} }
295   #4

```

```

296     { \tl_set:Nv #4 {#4} \prg_return_true: }
297     { \prg_return_false: }
298 }

```

3.9 Graph Conditionals

An expandable test for the existence of a vertex:

```

299 \prg_new_conditional:Nnn \graph_if_vertex_exist:Nn
300 {p, T, F, TF}
301 {
302   \prop_if_in:cnTF
303   { \__graph_tl:nnn {graph} {#1} {vertices} }
304   { #2 }
305   { \prg_return_true: }
306   { \prg_return_false: }
307 }

```

An expandable test for the existence of an edge:

```

308 \prg_new_conditional:Nnn \graph_if_edge_exist:Nnn
309 {p, T, F, TF}
310 {
311   \prop_if_in:coTF
312   { \__graph_tl:nnn {graph} {#1} {edge-values} }
313   { \__graph_key:nn{#2}{#3} }
314   { \prg_return_true: }
315   { \prg_return_false: }
316 }

```

Test whether graph #1 contains a cycle reachable from vertex #2:

```

317 \cs_new:Npn \graph_if_vertex_can_reach_cycle_p:Nn #1#2
318 { \__graph_if_vertex_can_reach_cycle_p:Nnn #1 {#2} {\__graph_empty_set} }
319 \cs_new:Npn \graph_if_vertex_can_reach_cycle:NnTF #1#2
320 { \__graph_if_vertex_can_reach_cycle:NnnTF #1 {#2} {\__graph_empty_set} }
321 \cs_new:Npn \graph_if_vertex_can_reach_cycle:NnT #1#2
322 { \__graph_if_vertex_can_reach_cycle:NnnT #1 {#2} {\__graph_empty_set} }
323 \cs_new:Npn \graph_if_vertex_can_reach_cycle:NnF #1#2
324 { \__graph_if_vertex_can_reach_cycle:NnnF #1 {#2} {\__graph_empty_set} }
325
326 \prg_new_conditional:Nnn \__graph_if_vertex_can_reach_cycle:Nnn
327 {p, T, F, TF}
328 % #1: graph id
329 % #2: vertex id
330 % #3: visited vertices in 'prop literal' format (internal l3prop)
331 {
332   \graph_map_outgoing_edges_tokens:Nnn #1 {#2}
333   { \__graph_if_vertex_can_reach_cycle:Nnnnn #1 {#3} }
334   \prg_return_false:
335 }
336
337 \cs_new:Nn \__graph_if_vertex_can_reach_cycle:Nnnnn
338 % #1: graph id
339 % #2: visited vertices in 'prop literal' format (internal l3prop)

```

```

340 % #3: start vertex (not used)
341 % #4: current vertex
342 % #5: edge value (behind ptr, not used)
343 {
344   \bool_if:nT
345   {
346     \__graph_set_if_in_p:nn {#2} {#4} ||
347     \__graph_if_vertex_can_reach_cycle_p:Nno #1 {#4}
348     { \__graph_set_cons:nn {#2} {#4} }
349   }
350   { \prop_map_break:n {\use_i:nn \prg_return_true:} }
351 }
352 \cs_generate_variant:Nn \__graph_if_vertex_can_reach_cycle_p:Nnn {Nno}

```

Test whether graph #1 contains any cycles:

```

353 \prg_new_conditional:Nnn \graph_if_cyclic:N
354 {p, T, F, TF}
355 % #1: graph id
356 {
357   \graph_map_vertices_tokens:Nn #1
358   { \__graph_if_cyclic:Nnn #1 }
359   \prg_return_false:
360 }
361
362 \cs_new:Nn \__graph_if_cyclic:Nnn
363 % #1: graph id
364 % #2: vertex id
365 % #3: vertex value (not used)
366 {
367   \bool_if:nT
368   { \graph_if_vertex_can_reach_cycle_p:Nn #1 {#2} }
369   { \prop_map_break:n {\use_i:nn \prg_return_true:} }
370 }

```

Test whether graph #1 contains any cycles:

```

371 % \prg_new_protected_conditional:Nnn \graph_get_cycle:NN
372 % {T, F, TF}
373 % % #1: graph id
374 % % #2: l3seq variable to put the cycle description in
375 % {
376 %   \seq_clear:N #2
377 %   \__graph_get_cycle:NNTF #1 #2
378 %   {\prg_return_true: }
379 %   {\prg_return_false:}
380 % }
381 %
382 % \prg_new_protected_conditional:Nnn \__graph_get_cycle:NN
383 % {T, F, TF}
384 % % #1: graph id
385 % % #2: l3seq variable
386 % {
387 %   \graph_map_successors_inline:Nnn #1 {} {

```

```

388 %      \seq_if_in:NnTF #2 {##1} {
389 %          % TODO
390 %      }{
391 %          % TODO
392 %      }
393 %  }
394 % }
395 %

```

Assume that graph #1 is acyclic and test whether a path exists from #2 to #3:

```

396 \prg_new_conditional:Nnn \graph_acyclic_if_path_exist:Nnn
397 {p, T, F, TF}
398 % #1: graph id
399 % #2: start vertex
400 % #3: end vertex
401 {
402   \graph_map_outgoing_edges_tokens:Nnn #1 {#2}
403   { \__graph_acyclic_if_path_exist:Nnnnn #1 {#3} }
404   \prg_return_false:
405 }
406
407 \cs_new:Nn \__graph_acyclic_if_path_exist:Nnnnn
408 % #1: graph id
409 % #2: end vertex
410 % #3: start vertex (not used)
411 % #4: possible end vertex
412 % #5: edge value (behind ptr, do not use)
413 {
414   \bool_if:nT
415   {
416     \str_if_eq_p:nn {#4} {#2} ||
417     \graph_acyclic_if_path_exist_p:Nnn #1 {#4} {#2}
418   }
419   { \prop_map_break:n {\use_i:nn \prg_return_true:} }
420 }

```

3.10 Querying Information

Get the number of vertices in the graph:

```

421 \cs_new:Nn \graph_vertex_count:N {
422   \int_use:c {\__graph_tl:nnn{graph}{#1}{vertex-count}}
423 }

```

Get the number of edges leading out of vertex #2:

```

424 \cs_new:Nn \graph_get_outdegree:Nn {
425   \prop_get:cn {\__graph_tl:nnn{graph}{#1}{outdegree}} {#2}
426 }

```

Get the number of edges leading into vertex #2:

```

427 \cs_new:Nn \graph_get_indegree:Nn {
428   \prop_get:cn { \__graph_tl:nnn{graph}{#1}{indegree}} {#2}
429 }

```

Get the number of edges connected to vertex #2:

```

430 \cs_new:Nn \graph_get_degree:Nn {
431   \int_eval:n{ \graph_get_outdegree:Nn #1 {#2} +
432               \graph_get_indegree:Nn #1 {#2} }
433 }

```

3.11 Mapping Graphs

Applies the tokens #2 to all vertex name/value pairs in the graph. The tokens are supplied with two arguments as trailing brace groups.

```

434 \cs_new:Nn \graph_map_vertices_tokens:Nn {
435   \prop_map_tokens:cn
436     { \__graph_tl:nnn{graph}{#1}{vertices} }
437     { \__graph_map_vertices_tokens_aux:nnv {#2} }
438 }
439 \cs_new:Nn \__graph_map_vertices_tokens_aux:nnn
440   { #1 {#2} {#3} }
441 \cs_generate_variant:Nn \__graph_map_vertices_tokens_aux:nnn {nnv}

```

Applies the function #2 to all vertex name/value pairs in the graph. The function is supplied with two arguments as trailing brace groups.

```

442 \cs_new:Nn \graph_map_vertices_function:NN {
443   \prop_map_tokens:cn
444     { \__graph_tl:nnn{graph}{#1}{vertices} }
445     { \exp_args:Nnv #2 }
446 }

```

Applies the inline function #2 to all vertex name/value pairs in the graph. The inline function is supplied with two arguments: ‘#1’ for the name, ‘#2’ for the value.

```

447 \cs_new_protected:Nn \graph_map_vertices_inline:Nn {
448   \withargs (c) [\uniquecsname] [#2] {
449     \cs_set:Npn ##1 #####1####2 {##2}
450     \graph_map_vertices_function:NN #1 ##1
451   }
452 }

```

Applies the tokens #2 to all edge from/to/value triples in the graph. The tokens are supplied with three arguments as trailing brace groups.

```

453 \cs_new:Nn \graph_map_edges_tokens:Nn {
454   \prop_map_tokens:cn
455     { \__graph_tl:nnn{graph}{#1}{edge-triples} }
456     { \__graph_map_edges_tokens_aux:nnn {#2} }
457 }
458 \cs_new:Nn \__graph_map_edges_tokens_aux:nnn
459   { \__graph_map_edges_tokens_aux:nnnv {#1} #3 }

```

```

460 \cs_new:Nn \__graph_map_edges_tokens_aux:nnnn
461 { #1 {#2} {#3} {#4} }
462 \cs_generate_variant:Nn \__graph_map_edges_tokens_aux:nnnn {nnnv}

```

Applies the function #2 to all edge from/to/value triples in the graph. The function is supplied with three arguments as trailing brace groups.

```

463 \cs_new:Nn \graph_map_edges_function:NN {
464   \prop_map_tokens:cn
465   { \__graph_tl:nnn{graph}{#1}{edge-triples} }
466   { \__graph_map_edges_function_aux:Nnn #2 }
467 }
468 \cs_new:Nn \__graph_map_edges_function_aux:Nnn
469 { \__graph_map_edges_function_aux:Nnnv #1 #3 }
470 \cs_new:Nn \__graph_map_edges_function_aux:Nnnn
471 { #1 {#2} {#3} {#4} }
472 \cs_generate_variant:Nn \__graph_map_edges_function_aux:Nnnn {Nnnv}

```

Applies the tokens #2 to all edge from/to/value triples in the graph. The tokens are supplied with three arguments: ‘#1’ for the ‘from’ vertex, ‘#2’ for the ‘to’ vertex and ‘#3’ for the edge value.

```

473 \cs_new_protected:Nn \graph_map_edges_inline:Nn {
474   \withargs (c) [\uniquecsname] [#2] {
475     \cs_set:Npn ##1 #####1####2####3 {##2}
476     \graph_map_edges_function:NN #1 ##1
477   }
478 }

```

Applies the tokens #3 to the from/to/value triples for the edges going ‘to’ vertex #2. The tokens are supplied with three arguments as trailing brace groups.

```

479 \cs_new:Nn \graph_map_incoming_edges_tokens:Nnn {
480   % #1: graph
481   % #2: base vertex
482   % #3: tokens to execute
483   \prop_map_tokens:cn
484   { \__graph_tl:nnn{graph}{#1}{edge-triples} }
485   { \__graph_map_incoming_edges_tokens_aux:nnnn {#2} {#3} }
486 }
487 \cs_new:Nn \__graph_map_incoming_edges_tokens_aux:nnnn
488 % #1: base vertex
489 % #2: tokens to execute
490 % #3: edge key
491 % #4: edge-triple {from}{to}{value}
492 { \__graph_map_incoming_edges_tokens_aux:nnnv {#1} {#2} #4 }
493 \cs_new:Nn \__graph_map_incoming_edges_tokens_aux:nnnnn
494 % #1: base vertex
495 % #2: tokens to execute
496 % #3: edge 'from' vertex
497 % #4: edge 'to' vertex
498 % #5: edge value

```



```

499 { \str_if_eq:nnT {#1} {#4} { #2 {#3} {#4} {#5} } }
500 \cs_generate_variant:Nn \__graph_map_incoming_edges_tokens_aux:nnnnn {nnnnv}

```

Applies the function #3 to the from/to/value triples for the edges going ‘to’ vertex #2. The function is supplied with three arguments as trailing brace groups.

```

501 \cs_new:Nn \graph_map_incoming_edges_function:NnN {
502   % #1: graph
503   % #2: base vertex
504   % #3: function to execute
505   \prop_map_tokens:cn
506   { \__graph_tl:nnn{graph}{#1}{edge-triples} }
507   { \__graph_map_incoming_edges_function_aux:nNnn {#2} #3 }
508 }
509 \cs_new:Nn \__graph_map_incoming_edges_function_aux:nNnn
510   % #1: base vertex
511   % #2: function to execute
512   % #3: edge key
513   % #4: edge-triple {from}{to}{value}
514   { \__graph_map_incoming_edges_function_aux:nNnnv {#1} #2 #4 }
515 \cs_new:Nn \__graph_map_incoming_edges_function_aux:nNnnn
516   % #1: base vertex
517   % #2: function to execute
518   % #3: edge ‘from’ vertex
519   % #4: edge ‘to’ vertex
520   % #5: edge value
521   { \str_if_eq:nnT {#1} {#4} { #2 {#3} {#4} {#5} } }
522 \cs_generate_variant:Nn \__graph_map_incoming_edges_function_aux:nNnnn {nNnnv}

```

Applies the inline function #3 to the from/to/value triples for the edges going ‘to’ vertex #2. The inline function is supplied with three arguments: ‘#1’ for the ‘from’ vertex, ‘#2’ is equal to the #2 supplied to this function and ‘#3’ contains the edge value.

```

523 \cs_new_protected:Nn \graph_map_incoming_edges_inline:Nnn {
524   % #1: graph
525   % #2: base vertex
526   % #3: body to execute
527   \withargs (c) [\uniquecsname] [#2] [#3] {
528     \cs_set:Npn ##1 #####2####3 {##3}
529     \graph_map_incoming_edges_function:NnN #1 {##2} ##1
530   }
531 }

```

Applies the tokens #3 to the from/to/value triples for the edges going ‘from’ vertex #2. The tokens are supplied with three arguments as trailing brace groups.

```

532 \cs_new:Nn \graph_map_outgoing_edges_tokens:Nnn {
533   % #1: graph
534   % #2: base vertex
535   % #3: tokens to execute
536   \prop_map_tokens:cn
537   { \__graph_tl:nnn{graph}{#1}{edge-triples} }
538   { \__graph_map_outgoing_edges_tokens_aux:nnnn {#2} {#3} }
539 }

```

```

540 \cs_new:Nn \__graph_map_outgoing_edges_tokens_aux:nnnn
541   % #1: base vertex
542   % #2: tokens to execute
543   % #3: edge key (not used)
544   % #4: edge-triple {from}{to}{value}
545   { \__graph_map_outgoing_edges_tokens_aux:nnnnv {#1} {#2} #4 }
546 \cs_new:Nn \__graph_map_outgoing_edges_tokens_aux:nnnnn
547   % #1: base vertex
548   % #2: tokens to execute
549   % #3: edge 'from' vertex
550   % #4: edge 'to' vertex
551   % #5: edge value
552   { \str_if_eq:nnT {#1} {#3} { #2 {#3} {#4} {#5} } }
553 \cs_generate_variant:Nn \__graph_map_outgoing_edges_tokens_aux:nnnnn {nnnnv}

```

Applies the function #3 to the from/to/value triples for the edges going ‘from’ vertex #2. The function is supplied with three arguments as trailing brace groups.

```

554 \cs_new:Nn \graph_map_outgoing_edges_function:NnN {
555   % #1: graph
556   % #2: base vertex
557   % #3: function to execute
558   \prop_map_tokens:cn
559     { \__graph_tl:nnn{graph}{#1}{edge-triples} }
560     { \__graph_map_outgoing_edges_function_aux:nNnn {#2} #3 }
561 }
562 \cs_new:Nn \__graph_map_outgoing_edges_function_aux:nNnn
563   % #1: base vertex
564   % #2: function to execute
565   % #3: edge key
566   % #4: edge-triple {from}{to}{value}
567   { \__graph_map_outgoing_edges_function_aux:nNnnv {#1} #2 #4 }
568 \cs_new:Nn \__graph_map_outgoing_edges_function_aux:nNnnn
569   % #1: base vertex
570   % #2: function to execute
571   % #3: edge 'from' vertex
572   % #4: edge 'to' vertex
573   % #5: edge value
574   { \str_if_eq:nnT {#1} {#3} { #2 {#3} {#4} {#5} } }
575 \cs_generate_variant:Nn \__graph_map_outgoing_edges_function_aux:nNnnn {nNnnv}

```

Applies the inline function #3 to the from/to/value triples for the edges going ‘from’ vertex #2. The inline function is supplied with three arguments: ‘#1’ is equal to the #2 supplied to this function, ‘#2’ contains the ‘to’ vertex and ‘#3’ contains the edge value.

```

576 \cs_new_protected:Nn \graph_map_outgoing_edges_inline:Nnn {
577   % #1: graph
578   % #2: base vertex
579   % #3: body to execute
580   \withargs (c) [\uniquecsname] [#2] [#3] {
581     \cs_set:Npn ##1 #####2#####3 {##3}
582     \graph_map_outgoing_edges_function:NnN #1 {##2} ##1

```

```

583 }
584 }

```

Applies the tokens #3 to the key/value pairs of the vertices reachable from vertex #2 in one step. The tokens are supplied with two arguments as trailing brace groups.

```

585 \cs_new:Nn \graph_map_successors_tokens:Nnn {
586   % #1: graph
587   % #2: base vertex
588   % #3: tokens to execute
589   \prop_map_tokens:cn
590     { \__graph_tl:nnn{graph}{#1}{edge-triples} }
591     { \__graph_map_successors_tokens_aux:Nnnnn #1 {#2} {#3} }
592 }
593 \cs_new:Nn \__graph_map_successors_tokens_aux:Nnnnn {
594   % #1: the graph
595   % #2: base vertex
596   % #3: tokens to execute
597   % #4: edge key (not used)
598   % #5: edge-triple {from}{to}{value}
599   \__graph_map_successors_tokens_aux:Nnnnnn #1 {#2} {#3} #5
600 }
601 \cs_new:Nn \__graph_map_successors_tokens_aux:Nnnnnn {
602   % #1: the graph
603   % #2: base vertex
604   % #3: tokens to execute
605   % #4: edge 'from' vertex
606   % #5: edge 'to' vertex
607   % #6: ptr to edge value (not used)
608   \str_if_eq:nnT {#2} {#4} {
609     \__graph_map_successors_tokens_aux:nnv
610       {#3} {#5} {\prop_get:cn{\__graph_tl:nnn{graph}{#1}{vertices}}{#5}}
611   }
612 }
613 \cs_new:Nn \__graph_map_successors_tokens_aux:nnn {
614   % #1: tokens to execute
615   % #2: successor key
616   % #3: successor value
617   #1 {#2} {#3}
618 }
619 \cs_generate_variant:Nn \__graph_map_successors_tokens_aux:nnn {nnv}

```

Applies the function #3 to the key/value pairs of the vertices reachable from vertex #2 in one step. The function is supplied with two arguments as trailing brace groups.

```

620 \cs_new:Nn \graph_map_successors_function:NnN {
621   % #1: graph
622   % #2: base vertex
623   % #3: function to execute
624   \prop_map_tokens:cn
625     { \__graph_tl:nnn{graph}{#1}{edge-triples} }
626     { \__graph_map_successors_function_aux:NnNnn #1 {#2} #3 }
627 }
628 \cs_new:Nn \__graph_map_successors_function_aux:NnNnn {

```

```

629 % #1: the graph
630 % #2: base vertex
631 % #3: function to execute
632 % #4: edge key (not used)
633 % #5: edge-triple {from}{to}{value}
634 \__graph_map_successors_function_aux:NnNnnn #1 {#2} #3 #5
635 }
636 \cs_new:Nn \__graph_map_successors_function_aux:NnNnnn {
637 % #1: the graph
638 % #2: base vertex
639 % #3: function to execute
640 % #4: edge 'from' vertex
641 % #5: edge 'to' vertex
642 % #6: ptr to edge value (not used)
643 \str_if_eq:nnT {#2} {#4} {
644 \__graph_map_successors_function_aux:Nnv
645 #3 {#5} {\prop_get:cn{\__graph_tl:nnn{graph}{#1}{vertices}}{#5}}
646 }
647 }
648 \cs_new:Nn \__graph_map_successors_function_aux:Nnn {
649 % #1: function to execute
650 % #2: successor key
651 % #3: successor value
652 #1 {#2} {#3}
653 }
654 \cs_generate_variant:Nn \__graph_map_successors_function_aux:Nnn {Nnv}

```

Applies the inline function #3 to the key/value pairs of the vertices reachable from vertex #2 in one step. The inline function is supplied with two arguments: ‘#1’ is the key, and ‘#2’ is the value of the successor vertex.

```

655 \cs_new_protected:Nn \graph_map_successors_inline:Nnn {
656 % #1: graph
657 % #2: base vertex
658 % #3: body to execute
659 \withargs (c) [\uniquecsname] [#2] [#3] {
660 \cs_set:Npn ##1 #####2####3 {##3}
661 \graph_map_successors_function:NnN #1 {##2} ##1
662 }
663 }

```

Applies the tokens #2 to all vertex name/value pairs in topological order. The tokens are supplied with two arguments as trailing brace groups. Assumes that the graph is acyclic (for now).

```

664 \cs_new_protected:Nn \graph_map_topological_order_tokens:Nn {
665
666 %% Fill \l__graph_source_vertices with source-nodes and count indegrees
667 %
668 \prop_gclear_new:c {l__graph_source_vertices_(\int_use:N\g__graph_nesting_depth_int)_prop}
669 \prop_gclear_new:c {l__graph_tmp_indeg_(\int_use:N\g__graph_nesting_depth_int)_prop}
670 \graph_map_vertices_inline:Nn #1 {
671 \prop_put:cnf {l__graph_tmp_indeg_(\int_use:N\g__graph_nesting_depth_int)_prop} {##1}
672 { \graph_get_indegree:Nn #1 {##1} }

```

```

673     \int_compare:nT {\graph_get_indegree:Nn #1 {##1} = 0} {
674         \prop_put:cnn {l__graph_source_vertices_(\int_use:N\g__graph_nesting_depth_int)_prop}
675     } }
676
677     %% Main loop
678     %
679     \bool_until_do:nn {\prop_if_empty_p:c {l__graph_source_vertices_(\int_use:N\g__graph_nesting_depth_int)_prop}
680         %% Choose any vertex (\l__graph_topo_key_tl, \l__graph_topo_value_tl)
681         %
682         \__graph_prop_any_key_pop:cN
683             {l__graph_source_vertices_(\int_use:N\g__graph_nesting_depth_int)_prop}
684             \l__graph_topo_key_tl
685         \graph_get_vertex:NVNT #1 \l__graph_topo_key_tl \l__graph_topo_val_tl {
686
687             %% Deduct one from the counter of all affected nodes
688             %% and add all now-empty vertices to source_vertices
689             %
690             \graph_map_outgoing_edges_inline:NVn #1 \l__graph_topo_key_tl {
691                 \prop_put:cnf {l__graph_tmp_indeg_(\int_use:N\g__graph_nesting_depth_int)_prop} {##2}
692                 {\int_eval:n {\prop_get:cn {l__graph_tmp_indeg_(\int_use:N\g__graph_nesting_depth_int)_prop} {##2}}
693                 \int_compare:nT {\prop_get:cn {l__graph_tmp_indeg_(\int_use:N\g__graph_nesting_depth_int)_prop} {##2} = 0}
694                 \prop_put:cnn {l__graph_source_vertices_(\int_use:N\g__graph_nesting_depth_int)_prop}
695             } }
696
697             %% Run the mapping function on the key and value from that vertex
698             %% and manage the nesting depth counter
699             %
700             \int_gincr:N \g__graph_nesting_depth_int
701             \withargs:VVn \l__graph_topo_key_tl \l__graph_topo_val_tl
702                 { #2 {##1} {##2} }
703             \int_gdecr:N \g__graph_nesting_depth_int
704         }
705     } }
706     \cs_new_protected:Nn \__graph_prop_any_key_pop:NN {
707         \prop_map_inline:Nn #1 {
708             \tl_set:Nn #2 {##1}
709             \prop_remove:Nn #1 {##1}
710             \prop_map_break:n {\use_none:nnn}
711         }
712         \tl_set:Nn #2 {\q_no_value} % TODO: test
713     }
714     \cs_generate_variant:Nn \__graph_prop_any_key_pop:NN {cN}
715     \cs_generate_variant:Nn \withargs:nnn {VVn}
716     \cs_generate_variant:Nn \graph_map_outgoing_edges_inline:Nnn {NVn}
717     \cs_generate_variant:Nn \prop_put:Nnn {cnf}
718     \cs_generate_variant:Nn \graph_get_vertex:NnNT {NVNT}
719     \tl_new:N \l__graph_topo_key_tl
720     \tl_new:N \l__graph_topo_val_tl
721     \int_new:N \g__graph_nesting_depth_int

```

Applies the function #2 to all vertex name/value pairs in topological order. The function is supplied with two arguments as trailing brace groups. Assumes that the graph is acyclic (for now).

```

722 \cs_new:Nn \graph_map_topological_order_function:NN {
723   \graph_map_topological_order_tokens:Nn #1 {#2}
724 }

```

Applies the inline function #2 to all vertex name/value pairs in topological order. The inline function is supplied with two arguments: ‘#1’ for the name and ‘#2’ for the value. Assumes that the graph is acyclic (for now).

```

725 \cs_new_protected:Nn \graph_map_topological_order_inline:Nn {
726   \withargs (c) [\uniquecsname] [#2] {
727     \cs_set:Npn ##1 #####1####2 {##2}
728     \graph_map_topological_order_function:NN #1 ##1
729   } }

```

3.12 Transforming Graphs

Set graph #1 to the transitive closure of graph #2.

```

730 \cs_new_protected:Nn \graph_set_transitive_closure:NN {
731   \__graph_set_transitive_closure:NNNnn #1 #2 \use_none:nnn {} { }
732 }
733 \cs_new_protected:Nn \graph_gset_transitive_closure:NN {
734   \__graph_set_transitive_closure:NNNnn #1 #2 \use_none:nnn {} {g}
735 }
736 \cs_new_protected:Nn \graph_set_transitive_closure:NNNn {
737   \__graph_set_transitive_closure:NNNnn #1 #2 #3 {#4} { }
738 }
739 \cs_new_protected:Nn \graph_gset_transitive_closure:NNNn {
740   \__graph_set_transitive_closure:NNNnn #1 #2 #3 {#4} {g}
741 }
742 \cs_new_protected:Nn \__graph_set_transitive_closure:NNNnn {
743   % #1: target
744   % #2: source
745   % #3: combination function with argspec :nnn
746   % #4: default 'old' value
747   \use:c{graph_#5set_eq:NN} #1 #2
748
749   \cs_set:Nn \__graph_edge_combinator:nnn {
750     \exp_not:n { #3 {##1} {##2} {##3} } }
751   \cs_generate_variant:Nn \__graph_edge_combinator:nnn {VVV}
752
753   \graph_map_vertices_inline:Nn #2 {
754     \graph_map_vertices_inline:Nn #2 {
755       \graph_get_edge:NnnNT #2 {##1} {####1}
756       \l__graph_edge_value_i_tl {
757         \graph_map_vertices_inline:Nn #2 {
758           \graph_get_edge:NnnNT #2 {####1} {#####1}
759           \l__graph_edge_value_ii_tl {
760             \graph_get_edge:NnnNF #1 {##1} {#####1}
761             \l__graph_edge_value_old_tl {
762               \tl_set:Nn \l__graph_edge_value_old_tl {#4}
763             }
764             \exp_args:NNx \tl_set:No \l__graph_edge_value_new_tl {
765               \__graph_edge_combinator:VVV

```

```

766         \l__graph_edge_value_i_tl
767         \l__graph_edge_value_ii_tl
768         \l__graph_edge_value_old_tl
769     }
770     \use:c{graph_#5put_edge:NnnV} #1 {##1} {#####1}
771     \l__graph_edge_value_new_tl
772 } } } } }
773 \cs_generate_variant:Nn \graph_put_edge:Nnnn {NnnV}
774 \cs_generate_variant:Nn \graph_gput_edge:Nnnn {NnnV}
775 \cs_generate_variant:Nn \tl_to_str:n {o}
776 \tl_new:N \l__graph_edge_value_i_tl
777 \tl_new:N \l__graph_edge_value_ii_tl
778 \tl_new:N \l__graph_edge_value_old_tl
779 \tl_new:N \l__graph_edge_value_new_tl

```

Assume that graph #2 contains no cycles, and set graph #1 to its transitive reduction.

```

780 \cs_new_protected:Nn \graph_set_transitive_reduction:NN {
781   \__graph_set_transitive_reduction:Nnn #1 #2 { } }
782 \cs_new_protected:Nn \graph_gset_transitive_reduction:NN {
783   \__graph_set_transitive_reduction:Nnn #1 #2 {g} }
784 \cs_new_protected:Nn \__graph_set_transitive_reduction:NNn {
785   % #1: target
786   % #2: source
787   \use:c{graph_#3set_eq:NN} #1 #2
788   \graph_map_vertices_inline:Nn #2 {
789     \graph_map_vertices_inline:Nn #2 {
790       \graph_get_edge:NnnNT #2 {##1} {#####1} \l_tmpa_tl {
791         \graph_map_vertices_inline:Nn #2 {
792           \graph_get_edge:NnnNT #2 {#####1} {#####1} \l_tmpa_tl {
793             \use:c{graph_#3remove_edge:Nnn} #1 {##1} {#####1}
794           } } } } }
795 }

```

3.13 Displaying Graphs

We define some additional functions that can display the graph in table-form. This is the option-less version, which delegates to the full version:

```

796 \cs_new_protected:Nn \graph_display_table:N {
797   \graph_display_table:Nn #1 { } }

```

The full version has a second argument accepting options that determine table formatting. We first define those options. Please note that with the standard options, the `xcolor` package is required with the `table` option, because of our use of the `\cellcolor` command.

```

798 \keys_define:nn {lt3graph-display} {
799   row_keys .bool_set:N = \l__graph_display_row_keys_bool,
800   row_keys .initial:n = {true},
801   row_keys .default:n = {true},
802
803   vertex_vals .bool_set:N = \l__graph_display_vertex_vals_bool,
804   vertex_vals .initial:n = {true},

```

```

805 vertex_vals .default:n = {true},
806
807 row_keys_format .tl_set:N = \l__graph_format_row_keys_tl,
808 row_keys_format .initial:n = \textbf,
809 row_keys_format .value_required:,
810
811 col_keys_format .tl_set:N = \l__graph_format_col_keys_tl,
812 col_keys_format .initial:n = \textbf,
813 col_keys_format .value_required:,
814
815 vertex_vals_format .tl_set:N = \l__graph_format_vertex_vals_tl,
816 vertex_vals_format .initial:n = \use:n,
817 vertex_vals_format .value_required:,
818
819 edge_vals_format .tl_set:N = \l__graph_format_edge_vals_tl,
820 edge_vals_format .initial:n = \use:n,
821 edge_vals_format .value_required:,
822
823 edge_diagonal_format .tl_set:N = \l__graph_format_edge_diagonal_tl,
824 edge_diagonal_format .initial:n = \cellcolor{black!30!white},
825 edge_diagonal_format .value_required:,
826
827 edge_direct_format .tl_set:N = \l__graph_format_edge_direct_tl,
828 edge_direct_format .initial:n = \cellcolor{green},
829 edge_direct_format .value_required:,
830
831 edge_transitive_format .tl_set:N = \l__graph_format_edge_transitive_tl,
832 edge_transitive_format .initial:n = \cellcolor{green!40!yellow}\tiny(tr),
833 edge_transitive_format .value_required:,
834
835 edge_none_format .tl_set:N = \l__graph_format_edge_none_tl,
836 edge_none_format .initial:n = {},
837 edge_none_format .value_required:
838 }

```

Now we define the function itself. It displays a table showing the structure and content of graph #1. If argument #2 is passed, its options are applied to format the output.

```

839 \cs_new_protected:Nn \graph_display_table:Nn {
840 \group_begin:

```

We process those options passed with #2:

```

841 \keys_set:nn {lt3graph-display} {#2}

```

We populate the top row of the table:

```

842 \tl_put_right:Nn \l__graph_table_content_tl {\hline}
843 \seq_clear:N \l__graph_row_seq
844 \bool_if:NT \l__graph_display_row_keys_bool
845 { \seq_put_right:Nn \l__graph_row_seq {}
846 \tl_put_right:Nn \l__graph_table_colspec_tl {|r|} }
847 \bool_if:NT \l__graph_display_vertex_vals_bool
848 { \seq_put_right:Nn \l__graph_row_seq {}

```



```

849     \tl_put_right:Nn \l__graph_table_colspec_tl {|c|} }
850 \graph_map_vertices_inline:Nn #1 {
851     \tl_put_right:Nn \l__graph_table_colspec_tl {|c}
852     \seq_put_right:Nn \l__graph_row_seq
853     { { \l__graph_format_col_keys_tl {##1} } }
854 }
855 \tl_put_right:Nn \l__graph_table_colspec_tl {|}
856 \tl_put_right:Nx \l__graph_table_content_tl
857     { \seq_use:Nn \l__graph_row_seq {&} }
858 \tl_put_right:Nn \l__graph_table_content_tl
859     { \\hline\hline }

```

We populate the remaining rows:

```

860 \graph_map_vertices_inline:Nn #1 {
861     \seq_clear:N \l__graph_row_seq
862     \bool_if:NT \l__graph_display_row_keys_bool {
863         \seq_put_right:Nn \l__graph_row_seq
864         { { \l__graph_format_row_keys_tl {##1} } } }
865     \bool_if:NT \l__graph_display_vertex_vals_bool {
866         \seq_put_right:Nn \l__graph_row_seq
867         { { \l__graph_format_vertex_vals_tl {##2} } } }
868     \graph_map_vertices_inline:Nn #1 {

```

We start building the vertex cell value. First we distinguish between a direct connection, a transitive connection, and no connection, and format accordingly:

```

869     \graph_get_edge:NnnNTF #1 {##1} {####1} \l_tmpa_tl {
870         \quark_if_no_value:VF \l_tmpa_tl {
871             \tl_set_eq:NN \l__graph_cell_content_tl \l_tmpa_tl
872             \tl_set:Nf \l__graph_cell_content_tl
873             { \exp_args:NV \l__graph_format_edge_direct_tl
874               \l__graph_cell_content_tl } }
875         }{\graph_acyclic_if_path_exist:NnnTF #1 {##1} {####1} {
876             \tl_set_eq:NN \l__graph_cell_content_tl
877             \l__graph_format_edge_transitive_tl
878         }{
879             \tl_set_eq:NN \l__graph_cell_content_tl
880             \l__graph_format_edge_none_tl
881         }}

```

Secondary formatting comes from cells on the diagonal, i.e., a key compared to itself:

```

882     \str_if_eq:nnT {##1} {####1} {
883         \tl_set:Nf \l__graph_cell_content_tl
884         { \exp_args:NV \l__graph_format_edge_diagonal_tl
885           \l__graph_cell_content_tl } }

```

Tertiary formatting is applied to all vertex value cells:

```

886     \tl_set:Nf \l__graph_cell_content_tl
887     { \exp_args:NV \l__graph_format_edge_vals_tl

```

```
888 \l__graph_cell_content_tl }
```

We can now add the cell to the row sequence:

```
889 \seq_put_right:NV \l__graph_row_seq \l__graph_cell_content_tl
890 }
```

We are finished with this row; go on to the next iteration:

```
891 \tl_put_right:Nx \l__graph_table_content_tl
892 { \seq_use:Nn \l__graph_row_seq {&} }
893 \tl_put_right:Nn \l__graph_table_content_tl {\\hline}
894 }
```

Finally, we print the table itself:

```
895 \withargs:VVn \l__graph_table_colspec_tl \l__graph_table_content_tl
896 { \begin{tabular}{##1}##2\end{tabular} }
897 \group_end:
898 }
```

Now follow the local variants and variables used in the function:

```
899 \cs_generate_variant:Nn \quark_if_no_value:nF {VF}
900 \cs_generate_variant:Nn \withargs:nnn {VVn}
901 \tl_new:N \l__graph_table_colspec_tl
902 \tl_new:N \l__graph_table_content_tl
903 \tl_new:N \l__graph_cell_content_tl
904 \bool_new:N \l__graph_table_skipfirst_bool
905 \seq_new:N \l__graph_row_seq
```

Change History

0.0.1		properly	1
	General: initial version	0.1.1	
0.0.8		General: fixed a similar bug in	
	General: a great many untracked	\graph(g)put_edges_from	1
	changes	0.1.2	
0.0.9		General: allowing \graph_map_topo-	
	General: creation of the documentation	logical_order... to be nested .	1
0.1.0		0.1.3	
	General: fixed a bug in	General: fixed a bug in \graph_re-	
	\graph(g)put_vertex and	move_vertex and added a \graph_-	
	\graph(g)put_edge, which caused	vertex_count:N function	1
	their global versions not to work		

Index

The italic numbers denote the pages where the corresponding entry is described, numbers underlined point to the definition, all others indicate the places where it is used.

Symbols	
\\	859, 893
_graph_acyclic_if_path_exist:Nnnnn	403, 407
_graph_clear:Nn	69, 71, 72
_graph_clear_new:Nn	77, 79, 80
_graph_edge_combinator:VVV	765
_graph_edge_combinator:nnn	749, 751
_graph_empty_set	21, 318, 320, 322, 324
_graph_for_each_prop_datatype:n	36, 63, 73, 91
_graph_get_cycle:NN	382
_graph_get_cycle:NNTF	377
_graph_gput_edges_from:NNn	269, 271, 272
_graph_if_cyclic:Nnn	358, 362
_graph_if_vertex_can_reach_cycle:Nnn	326
_graph_if_vertex_can_reach_cycle:NnnF	324
_graph_if_vertex_can_reach_cycle:NnnT	322
_graph_if_vertex_can_reach_cycle:NnnTF	320
_graph_if_vertex_can_reach_cycle:Nnnnn	333, 337
_graph_if_vertex_can_reach_cycle_p:Nnn	318, 352
_graph_if_vertex_can_reach_cycle_p:Nno	347
_graph_key:n	31
_graph_key:nn	32, 166, 170, 175, 179, 251, 254, 257, 260, 294, 313
_graph_key:nnn	33
_graph_key:nnnn	34
_graph_key:nnnnn	35
_graph_map_edges_function_aux:Nnn	466, 468
_graph_map_edges_function_aux:Nnnn	470, 472
_graph_map_edges_function_aux:Nnnv	469
_graph_map_edges_tokens_aux:nnn	456, 458
_graph_map_edges_tokens_aux:nnnn	460, 462
_graph_map_edges_tokens_aux:nnnv	459
_graph_map_incoming_edges_function_aux:nNnn	507, 509
_graph_map_incoming_edges_function_aux:nNnnn	515, 522
_graph_map_incoming_edges_function_aux:nNnnv	514
_graph_map_incoming_edges_tokens_aux:nnnn	485, 487
_graph_map_incoming_edges_tokens_aux:nnnnn	493, 500
_graph_map_incoming_edges_tokens_aux:nnnnv	492
_graph_map_outgoing_edges_function_aux:nNnn	560, 562
_graph_map_outgoing_edges_function_aux:nNnnn	568, 575
_graph_map_outgoing_edges_function_aux:nNnnv	567
_graph_map_outgoing_edges_tokens_aux:nnnn	538, 540
_graph_map_outgoing_edges_tokens_aux:nnnnn	546, 553
_graph_map_outgoing_edges_tokens_aux:nnnnv	545
_graph_map_successors_function_aux:NnNnn	626, 628
_graph_map_successors_function_aux:NnNnnn	634, 636
_graph_map_successors_function_aux:Nnn	648, 654
_graph_map_successors_function_aux:Nnv	644
_graph_map_successors_tokens_aux:Nnnnnn	591, 593
_graph_map_successors_tokens_aux:Nnnnnn	599, 601
_graph_map_successors_tokens_aux:nnn	613, 619
_graph_map_successors_tokens_aux:nnv	609
_graph_map_vertices_tokens_aux:nnn	439, 441
_graph_map_vertices_tokens_aux:nnv	437
_graph_prop_any_key_pop:NN	706, 714
_graph_prop_any_key_pop:cN	682
_graph_ptr_gnew:Nn	49
_graph_ptr_new:Nn	42

\l__graph_format_edge_vals_tl	819, 887		
\l__graph_format_row_keys_tl	807, 864		
\l__graph_format_vertex_vals_tl	815, 867		
\l__graph_from_value_tl	143, 163, 195		
\l__graph_row_seq	843, 845, 848, 852, 857, 861, 863, 866, 889, 892, 905		
\l__graph_source_vertices	666		
\l__graph_table_colspec_tl	846, 849, 851, 855, 895, 901		
\l__graph_table_content_tl	842, 856, 858, 891, 893, 895, 902		
\l__graph_table_skipfirst_bool	904		
\l__graph_to_value_tl	144, 163, 196		
\l__graph_topo_key_tl	680, 684, 685, 690, 701, 719		
\l__graph_topo_val_tl	685, 701, 720		
\l__graph_topo_value_tl	680		
\l__graph_vertex_data_tl	114, 119, 132, 203, 226		
\l_tmpa_tl	125, 145, 790, 792, 869, 870, 871		
		N	
\NeedsTeXFormat	1		
		P	
\prg_new_conditional:Nnn	299, 308, 326, 353, 396		
\prg_new_conditional:Npnn	13		
\prg_new_protected_conditional:Nnn	282, 289, 371, 382		
\prg_return_false:	287, 297, 306, 315, 334, 359, 379, 404		
\prg_return_true:	286, 296, 305, 314, 350, 369, 378, 419		
\prop_gclear_new:c	668, 669		
\prop_get:cn	150, 157, 237, 244, 425, 428, 610, 645, 692, 693		
\prop_get:cnNTF	285		
\prop_get:coNTF	292		
\prop_gput:Nnn	191, 266		
\prop_gremove:Nn	264		
\prop_if_empty_p:c	679		
\prop_if_in:cnTF	302		
\prop_if_in:coTF	311		
\prop_map_break:n	350, 369, 419, 710		
\prop_map_inline:Nn	707		
\prop_map_tokens:cn	435, 443, 454, 464, 483, 505, 536, 558, 589, 624		
\prop_new:c	64		
\prop_put:cnf	671, 691		
\prop_put:cnn	674, 694		
\prop_put:Nnn	192, 225, 265, 717		
\prop_remove:Nn	263, 709		
\ProvidesExplPackage	3		
		Q	
\q_no_value	712		
\q_recursion_tail	17		
\quark_if_no_value:nF	899		
\quark_if_no_value:VF	870		
		R	
\RequirePackage	2, 5, 6, 7, 8, 9, 10, 11, 12		
		S	
\s__prop	16, 21, 24		
\s_obj_end	15		
\seq_clear:N	376, 843, 861		
\seq_if_in:NnTF	388		
\seq_map_inline:Nn	37		
\seq_new:N	38, 905		
\seq_put_right:Nn	845, 848, 852, 863, 866		
\seq_put_right:NV	889		
\seq_set_from_clist:Nn	39		
\seq_use:Nn	857, 892		
\str_if_eq:nnT	499, 521, 552, 574, 608, 643, 882		
\str_if_eq_p:nn	416		
\str_tail:n	61		
		T	
\textbf	808, 812		
\tiny	832		
\tl_gset:cn	53		
\tl_gset:Nn	51		
\tl_new:c	45, 52		
\tl_new:N	60, 132, 194, 195, 196, 226, 267, 719, 720, 776, 777, 778, 779, 901, 902, 903		
\tl_put_right:Nn	842, 846, 849, 851, 855, 858, 893		
\tl_put_right:Nx	856, 891		
\tl_set:cn	46		
\tl_set:Nf	61, 872, 883, 886		
\tl_set:Nn	44, 708, 712, 762		
\tl_set:No	764		
\tl_set:Nv	286, 296		
\tl_set_eq:NN	871, 876, 879		
\tl_to_str:n	167, 171, 180, 181, 775		
\tl_trim_spaces:f	61		
\tl_trim_spaces:n	67		
		U	
\uniquecsname	43, 50, 448, 474, 527, 580, 659, 726		
\use:c	74, 83, 90, 93, 114, 118, 123, 128, 129, 148, 155, 164, 168, 172, 173, 177, 207, 212, 216, 217, 218, 222, 235, 242, 249, 252, 255, 258, 277, 747, 770, 787, 793		

<code>\use:n</code>	816, 820	W
<code>\use_i:nn</code>	350, 369, 419	<code>\withargs</code> 43, 50, 448, 474, 527, 580, 659, 726
<code>\use_none:nnn</code>	710, 731, 734	<code>\withargs:nnn</code> 193, 715, 900
		<code>\withargs:VVn</code> 163, 701, 895