

The `lt3graph` package^{*†}

Michiel Helvensteijn
mhelvans+latex@gmail.com

April 13, 2014

Development of this package is organized at github.com/mhelvans/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` v0.0.9-r2, dated 2013/12/05.

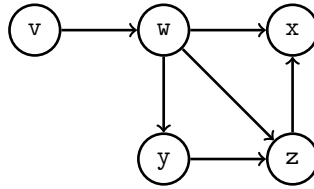
[†]The prefix `lt3` indicates that this package is a user-contributed `expl3` library, in contrast to packages prefixed with `l3`, which are officially supported by the $\text{\LaTeX}3$ team.

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}{2013/12/05}{0.0.9-r2}
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     \__graph_for_each_prop_datatype:n
63       { \prop_new:c {\__graph_tl:nnn{graph}{#1}{##1}} }
64   }
65 }

```

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

Remove all data from a graph:

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

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

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

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

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

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)`:

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

3.7 Manipulating graphs

Put a new vertex inside a graph:

```

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

```

Put a new edge inside a graph:

```

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

```



```

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

```

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

```

192 \cs_new_protected:Nn \graph_remove_vertex:Nn
193 { \__graph_remove_vertex:Nnn #1 {#2} { } }
194 \cs_new_protected:Nn \graph_gremove_vertex:Nn
195 { \__graph_remove_vertex:Nnn #1 {#2} {g} }
196 \cs_new_protected:Nn \__graph_remove_vertex:Nnn
197 {
198     \graph_get_vertex:NnNT #1 {#2} \l__graph_vertex_data_tl {
199         %%% remove outgoing edges
200         %
201         \graph_map_outgoing_edges_inline:Nnn #1 {#2}

```

```

202     { \use:c{graph_#3remove_edge:Nnn} #1 {##1} {##2} }
203
204     %%% remove incoming edges
205     %
206     \graph_map_incoming_edges_inline:Nnn #1 {#2}
207     { \use:c{graph_#3remove_edge:Nnn} #1 {##1} {##2} }
208
209     %%% remove the vertex
210     %
211     \use{prop_#3remove:cn} {\__graph_tl:nnn{graph}{#1}{vertices}} {#2}
212     \use{prop_#3remove:cn} {\__graph_tl:nnn{graph}{#1}{indegree}} {#2}
213     \use{prop_#3remove:cn} {\__graph_tl:nnn{graph}{#1}{outdegree}} {#2}
214   }
215 }
216 \cs_generate_variant:Nn \prop_put:Nnn {cnV}
217 % \tl_new:N \l__graph_vertex_data_tl % reusing from other function

```

Remove an edge from the graph:

```

218 \cs_new_protected:Nn \graph_remove_edge:Nnn
219 { \__graph_remove_edge:Nnnn #1 {#2} {#3} { } }
220 \cs_new_protected:Nn \graph_gremove_edge:Nnn
221 { \__graph_remove_edge:Nnnn #1 {#2} {#3} {g} }
222 \cs_new_protected:Nn \__graph_remove_edge:Nnnn {
223   \graph_get_edge:NnnNT #1 {#2} {#3} \l__graph_edge_data_tl {
224     %%% decrement outdegree of vertex #2
225     %
226     \use:c{prop_#4put:cnf} {\__graph_tl:nnn{graph}{#1}{outdegree}} {#2}
227     {\int_eval:n {
228       \prop_get:cn {\__graph_tl:nnn{graph}{#1}{outdegree}} {#2} - 1
229     }}
230
231     %%% decrement indegree of vertex #3
232     %
233     \use:c{prop_#4put:cnf} {\__graph_tl:nnn{graph}{#1}{indegree}} {#3}
234     {\int_eval:n {
235       \prop_get:cn {\__graph_tl:nnn{graph}{#1}{indegree}} {#3} - 1
236     }}
237
238     %%% actually remove edge
239     %
240     \use:c{prop_#4remove:co}
241     { \__graph_tl:nnn{graph}{#1}{edge-froms} }
242     { \__graph_key:nn{#2}{#3} }
243     \use:c{prop_#4remove:co}
244     { \__graph_tl:nnn{graph}{#1}{edge-tos} }
245     { \__graph_key:nn{#2}{#3} }
246     \use:c{prop_#4remove:co}
247     { \__graph_tl:nnn{graph}{#1}{edge-values} }
248     { \__graph_key:nn{#2}{#3} }
249     \use:c{prop_#4remove:co}
250     { \__graph_tl:nnn{graph}{#1}{edge-triples} }
251     { \__graph_key:nn{#2}{#3} }
252   }

```

```

253 }
254 \cs_generate_variant:Nn \prop_remove:Nn {co}
255 \cs_generate_variant:Nn \prop_gremove:Nn {co}
256 \cs_generate_variant:Nn \prop_put:Nnn {cnf}
257 \cs_generate_variant:Nn \prop_gput:Nnn {cnf}
258 %\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:

```

259 \cs_new_protected:Nn \graph_put_edges_from:NN
260 { \__graph_gput_edges_from:Nnn #1 #2 { } }
261 \cs_new_protected:Nn \graph_gput_edges_from:NN
262 { \__graph_gput_edges_from:Nnn #1 #2 {g} }
263 \cs_new_protected:Nn \__graph_gput_edges_from:Nnn
264 {
265   \graph_map_edges_inline:Nn #2 {
266     \graph_if_vertex_exist:NnT #1 {##1} {
267       \graph_if_vertex_exist:NnT #1 {##2} {
268         \graph_gput_edge:Nnnn #1 {##1} {##2} {##3}
269       }
270     }
271   }
272 }

```

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.

```

273 \prg_new_protected_conditional:Nnn \graph_get_vertex:NnN
274 {T, F, TF}
275 {
276   \prop_get:cnNTF { \__graph_tl:nnn {graph} {#1} {vertices} } {#2} #3
277   { \tl_set:Nv #3 {#3} \prg_return_true: }
278   { \prg_return_false: }
279 }

```

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.

```

280 \prg_new_protected_conditional:Nnn \graph_get_edge:NnnN
281 {T, F, TF}
282 {
283   \prop_get:cnNTF
284   { \__graph_tl:nnn{graph}{#1}{edge-values} }
285   { \__graph_key:nn{#2}{#3} }
286   #4
287   { \tl_set:Nv #4 {#4} \prg_return_true: }
288   { \prg_return_false: }
289 }

```

3.9 Graph Conditionals

An expandable test for the existence of a vertex:

```

290 \prg_new_conditional:Nnn \graph_if_vertex_exist:Nn
291 {p, T, F, TF}
292 {
293   \prop_if_in:cnTF
294   { \__graph_tl:nnn {graph} {#1} {vertices} }
295   { #2 }
296   { \prg_return_true: }
297   { \prg_return_false: }
298 }

```

An expandable test for the existence of an edge:

```

299 \prg_new_conditional:Nnn \graph_if_edge_exist:Nnn
300 {p, T, F, TF}
301 {
302   \prop_if_in:coTF
303   { \__graph_tl:nnn {graph} {#1} {edge-values} }
304   { \__graph_key:nn{#2}{#3} }
305   { \prg_return_true: }
306   { \prg_return_false: }
307 }

```

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

```

308 \cs_new:Npn \graph_if_vertex_can_reach_cycle_p:Nn #1#2
309 { \__graph_if_vertex_can_reach_cycle_p:Nnn #1 {#2} {\__graph_empty_set} }
310 \cs_new:Npn \graph_if_vertex_can_reach_cycle:NnTF #1#2
311 { \__graph_if_vertex_can_reach_cycle:NnnTF #1 {#2} {\__graph_empty_set} }
312 \cs_new:Npn \graph_if_vertex_can_reach_cycle:NnT #1#2
313 { \__graph_if_vertex_can_reach_cycle:NnnT #1 {#2} {\__graph_empty_set} }
314 \cs_new:Npn \graph_if_vertex_can_reach_cycle:NnF #1#2
315 { \__graph_if_vertex_can_reach_cycle:NnnF #1 {#2} {\__graph_empty_set} }
316
317 \prg_new_conditional:Nnn \__graph_if_vertex_can_reach_cycle:Nnn
318 {p, T, F, TF}
319 % #1: graph id
320 % #2: vertex id
321 % #3: visited vertices in 'prop literal' format (internal l3prop)
322 {
323   \graph_map_outgoing_edges_tokens:Nnn #1 {#2}
324   { \__graph_if_vertex_can_reach_cycle:Nnnnn #1 {#3} }
325   \prg_return_false:
326 }
327
328 \cs_new:Nn \__graph_if_vertex_can_reach_cycle:Nnnnn
329 % #1: graph id
330 % #2: visited vertices in 'prop literal' format (internal l3prop)
331 % #3: start vertex (not used)
332 % #4: current vertex
333 % #5: edge value (behind ptr, not used)
334 {
335   \bool_if:nT
336   {
337     \__graph_set_if_in_p:nn {#2} {#4} ||

```

```

338     \_graph_if_vertex_can_reach_cycle_p:Nno #1 {#4}
339     { \_graph_set_cons:nn {#2} {#4} }
340   }
341   { \prop_map_break:n {\use_i:nn \prg_return_true:} }
342 }
343 \cs_generate_variant:Nn \_graph_if_vertex_can_reach_cycle_p:Nnn {Nno}

```

Test whether graph #1 contains any cycles:

```

344 \prg_new_conditional:Nnn \graph_if_cyclic:N
345 {p, T, F, TF}
346 % #1: graph id
347 {
348   \graph_map_vertices_tokens:Nn #1
349   { \_graph_if_cyclic:Nnn #1 }
350   \prg_return_false:
351 }
352
353 \cs_new:Nn \_graph_if_cyclic:Nnn
354 % #1: graph id
355 % #2: vertex id
356 % #3: vertex value (not used)
357 {
358   \bool_if:nT
359   { \graph_if_vertex_can_reach_cycle_p:Nn #1 {#2} }
360   { \prop_map_break:n {\use_i:nn \prg_return_true:} }
361 }

```

Test whether graph #1 contains any cycles:

```

362 % \prg_new_protected_conditional:Nnn \graph_get_cycle:NN
363 % {T, F, TF}
364 % % #1: graph id
365 % % #2: l3seq variable to put the cycle description in
366 % {
367 %   \seq_clear:N #2
368 %   \_graph_get_cycle:NNTF #1 #2
369 %   {\prg_return_true: }
370 %   {\prg_return_false:}
371 % }
372 %
373 % \prg_new_protected_conditional:Nnn \_graph_get_cycle:NN
374 % {T, F, TF}
375 % % #1: graph id
376 % % #2: l3seq variable
377 % {
378 %   \graph_map_successors_inline:Nnn #1 {} {
379 %     \seq_if_in:NnTF #2 {##1} {
380 %       % TODO
381 %     }{
382 %       % TODO
383 %     }
384 %   }
385 % }

```

```
386 %
```

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

```
387 \prg_new_conditional:Nnn \graph_acyclic_if_path_exist:Nnn
388 {p, T, F, TF}
389 % #1: graph id
390 % #2: start vertex
391 % #3: end vertex
392 {
393   \graph_map_outgoing_edges_tokens:Nnn #1 {#2}
394   { \__graph_acyclic_if_path_exist:Nnnnn #1 {#3} }
395   \prg_return_false:
396 }
397
398 \cs_new:Nn \__graph_acyclic_if_path_exist:Nnnnn
399 % #1: graph id
400 % #2: end vertex
401 % #3: start vertex (not used)
402 % #4: possible end vertex
403 % #5: edge value (behind ptr, do not use)
404 {
405   \bool_if:nT
406   {
407     \str_if_eq_p:nn {#4} {#2} ||
408     \graph_acyclic_if_path_exist_p:Nnn #1 {#4} {#2}
409   }
410   { \prop_map_break:n {\use_i:nn \prg_return_true:} }
411 }
```

3.10 Querying Information

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

```
412 \cs_new:Nn \graph_get_outdegree:Nn {
413   \prop_get:cn {\__graph_tl:nnn{graph}{#1}{outdegree}} {#2}
414 }
```

Get the number of edges leading into vertex #2:

```
415 \cs_new:Nn \graph_get_indegree:Nn {
416   \prop_get:cn {\__graph_tl:nnn{graph}{#1}{indegree}} {#2}
417 }
```

Get the number of edges connected to vertex #2:

```
418 \cs_new:Nn \graph_get_degree:Nn {
419   \int_eval:n{ \graph_get_outdegree:Nn #1 {#2} +
420               \graph_get_indegree:Nn #1 {#2} }
421 }
```

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.

```

422 \cs_new:Nn \graph_map_vertices_tokens:Nn {
423   \prop_map_tokens:cn
424   { \__graph_tl:nnn{graph}{#1}{vertices} }
425   { \__graph_map_vertices_tokens_aux:nnv {#2} }
426 }
427 \cs_new:Nn \__graph_map_vertices_tokens_aux:nnn
428   { #1 {#2} {#3} }
429 \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.

```

430 \cs_new:Nn \graph_map_vertices_function:NN {
431   \prop_map_tokens:cn
432   { \__graph_tl:nnn{graph}{#1}{vertices} }
433   { \exp_args:Nnv #2 }
434 }

```

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.

```

435 \cs_new_protected:Nn \graph_map_vertices_inline:Nn {
436   \withargs (c) [\uniquecsname] [#2] {
437     \cs_set:Npn ##1 #####1####2 {##2}
438     \graph_map_vertices_function:NN #1 #1
439   }
440 }

```

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.

```

441 \cs_new:Nn \graph_map_edges_tokens:Nn {
442   \prop_map_tokens:cn
443   { \__graph_tl:nnn{graph}{#1}{edge-triples} }
444   { \__graph_map_edges_tokens_aux:nnn {#2} }
445 }
446 \cs_new:Nn \__graph_map_edges_tokens_aux:nnn
447   { \__graph_map_edges_tokens_aux:nnnv {#1} #3 }
448 \cs_new:Nn \__graph_map_edges_tokens_aux:nnnn
449   { #1 {#2} {#3} {#4} }
450 \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.

```

451 \cs_new:Nn \graph_map_edges_function:NN {
452   \prop_map_tokens:cn
453   { \__graph_tl:nnn{graph}{#1}{edge-triples} }
454   { \__graph_map_edges_function_aux:Nnn #2 }
455 }
456 \cs_new:Nn \__graph_map_edges_function_aux:Nnn
457   { \__graph_map_edges_function_aux:nnnv #1 #3 }
458 \cs_new:Nn \__graph_map_edges_function_aux:Nnnn
459   { #1 {#2} {#3} {#4} }

```

```
460 \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.

```
461 \cs_new_protected:Nn \graph_map_edges_inline:Nn {
462   \withargs (c) [\uniquecsname] [#2] {
463     \cs_set:Npn ##1 #####1####2####3 {##2}
464     \graph_map_edges_function:NN #1 ##1
465   }
466 }
```

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.

```
467 \cs_new:Nn \graph_map_incoming_edges_tokens:Nnn {
468   % #1: graph
469   % #2: base vertex
470   % #3: tokens to execute
471   \prop_map_tokens:cn
472   { \__graph_tl:nnn{graph}{#1}{edge-triples} }
473   { \__graph_map_incoming_edges_tokens_aux:nnnn {#2} {#3} }
474 }
475 \cs_new:Nn \__graph_map_incoming_edges_tokens_aux:nnnn
476   % #1: base vertex
477   % #2: tokens to execute
478   % #3: edge key
479   % #4: edge-triple {from}{to}{value}
480   { \__graph_map_incoming_edges_tokens_aux:nnnnv {#1} {#2} #4 }
481 \cs_new:Nn \__graph_map_incoming_edges_tokens_aux:nnnnn
482   % #1: base vertex
483   % #2: tokens to execute
484   % #3: edge ‘from’ vertex
485   % #4: edge ‘to’ vertex
486   % #5: edge value
487   { \str_if_eq:nnT {#1} {#4} { #2 {#3} {#4} {#5} } }
488 \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.

```
489 \cs_new:Nn \graph_map_incoming_edges_function:NnN {
490   % #1: graph
491   % #2: base vertex
492   % #3: function to execute
493   \prop_map_tokens:cn
494   { \__graph_tl:nnn{graph}{#1}{edge-triples} }
495   { \__graph_map_incoming_edges_function_aux:nNnn {#2} #3 }
496 }
497 \cs_new:Nn \__graph_map_incoming_edges_function_aux:nNnn
498   % #1: base vertex
499   % #2: function to execute
500   % #3: edge key
```



```

501 % #4: edge-triple {from}{to}{value}
502 { \__graph_map_incoming_edges_function_aux:nNnnv {#1} #2 #4 }
503 \cs_new:Nn \__graph_map_incoming_edges_function_aux:nNnnn
504 % #1: base vertex
505 % #2: function to execute
506 % #3: edge 'from' vertex
507 % #4: edge 'to' vertex
508 % #5: edge value
509 { \str_if_eq:nnT {#1} {#4} { #2 {#3} {#4} {#5} } }
510 \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.

```

511 \cs_new_protected:Nn \graph_map_incoming_edges_inline:Nnn {
512 % #1: graph
513 % #2: base vertex
514 % #3: body to execute
515 \withargs (c) [\uniquecsname] [#2] [#3] {
516   \cs_set:Npn ##1 #####2####3 {##3}
517   \graph_map_incoming_edges_function:NnN #1 {##2} ##1
518 }
519 }

```

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.

```

520 \cs_new:Nn \graph_map_outgoing_edges_tokens:Nnn {
521 % #1: graph
522 % #2: base vertex
523 % #3: tokens to execute
524 \prop_map_tokens:cn
525 { \__graph_tl:nnn{graph}{#1}{edge-triples} }
526 { \__graph_map_outgoing_edges_tokens_aux:nnnn {#2} {#3} }
527 }
528 \cs_new:Nn \__graph_map_outgoing_edges_tokens_aux:nnnn
529 % #1: base vertex
530 % #2: tokens to execute
531 % #3: edge key (not used)
532 % #4: edge-triple {from}{to}{value}
533 { \__graph_map_outgoing_edges_tokens_aux:nnnnv {#1} {#2} #4 }
534 \cs_new:Nn \__graph_map_outgoing_edges_tokens_aux:nnnnn
535 % #1: base vertex
536 % #2: tokens to execute
537 % #3: edge 'from' vertex
538 % #4: edge 'to' vertex
539 % #5: edge value
540 { \str_if_eq:nnT {#1} {#3} { #2 {#3} {#4} {#5} } }
541 \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.

```

542 \cs_new:Nn \graph_map_outgoing_edges_function:NnN {
543   % #1: graph
544   % #2: base vertex
545   % #3: function to execute
546   \prop_map_tokens:cn
547     { \__graph_tl:nnn{graph}{#1}{edge-triples} }
548     { \__graph_map_outgoing_edges_function_aux:nNnn {#2} #3 }
549 }
550 \cs_new:Nn \__graph_map_outgoing_edges_function_aux:nNnn
551   % #1: base vertex
552   % #2: function to execute
553   % #3: edge key
554   % #4: edge-triple {from}{to}{value}
555   { \__graph_map_outgoing_edges_function_aux:nNnnv {#1} #2 #4 }
556 \cs_new:Nn \__graph_map_outgoing_edges_function_aux:nNnnn
557   % #1: base vertex
558   % #2: function to execute
559   % #3: edge 'from' vertex
560   % #4: edge 'to' vertex
561   % #5: edge value
562   { \str_if_eq:nnT {#1} {#3} { #2 {#3} {#4} {#5} } }
563 \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.

```

564 \cs_new_protected:Nn \graph_map_outgoing_edges_inline:Nnn {
565   % #1: graph
566   % #2: base vertex
567   % #3: body to execute
568   \withargs (c) [\uniquecsname] [#2] [#3] {
569     \cs_set:Npn ##1 #####2####3 {##3}
570     \graph_map_outgoing_edges_function:NnN #1 {##2} ##1
571   }
572 }

```

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.

```

573 \cs_new:Nn \graph_map_successors_tokens:Nnn {
574   % #1: graph
575   % #2: base vertex
576   % #3: tokens to execute
577   \prop_map_tokens:cn
578     { \__graph_tl:nnn{graph}{#1}{edge-triples} }
579     { \__graph_map_successors_tokens_aux:Nnnnn #1 {#2} {#3} }
580 }
581 \cs_new:Nn \__graph_map_successors_tokens_aux:Nnnnn {
582   % #1: the graph
583   % #2: base vertex
584   % #3: tokens to execute
585   % #4: edge key (not used)

```

```

586 % #5: edge-triple {from}{to}{value}
587 \__graph_map_successors_tokens_aux:Nnnnnn #1 {#2} {#3} #5
588 }
589 \cs_new:Nn \__graph_map_successors_tokens_aux:Nnnnnn {
590 % #1: the graph
591 % #2: base vertex
592 % #3: tokens to execute
593 % #4: edge 'from' vertex
594 % #5: edge 'to' vertex
595 % #6: ptr to edge value (not used)
596 \str_if_eq:nnT {#2} {#4} {
597   \__graph_map_successors_tokens_aux:nnv
598     {#3} {#5} {\prop_get:cn{\__graph_tl:nnn{graph}{#1}{vertices}}{#5}}
599 }
600 }
601 \cs_new:Nn \__graph_map_successors_tokens_aux:nnn {
602 % #1: tokens to execute
603 % #2: successor key
604 % #3: successor value
605 #1 {#2} {#3}
606 }
607 \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.

```

608 \cs_new:Nn \graph_map_successors_function:NnN {
609 % #1: graph
610 % #2: base vertex
611 % #3: function to execute
612 \prop_map_tokens:cn
613   { \__graph_tl:nnn{graph}{#1}{edge-triples} }
614   { \__graph_map_successors_function_aux:NnNnn #1 {#2} #3 }
615 }
616 \cs_new:Nn \__graph_map_successors_function_aux:NnNnn {
617 % #1: the graph
618 % #2: base vertex
619 % #3: function to execute
620 % #4: edge key (not used)
621 % #5: edge-triple {from}{to}{value}
622 \__graph_map_successors_function_aux:NnNnnn #1 {#2} #3 #5
623 }
624 \cs_new:Nn \__graph_map_successors_function_aux:NnNnnn {
625 % #1: the graph
626 % #2: base vertex
627 % #3: function to execute
628 % #4: edge 'from' vertex
629 % #5: edge 'to' vertex
630 % #6: ptr to edge value (not used)
631 \str_if_eq:nnT {#2} {#4} {
632   \__graph_map_successors_function_aux:Nnv
633     #3 {#5} {\prop_get:cn{\__graph_tl:nnn{graph}{#1}{vertices}}{#5}}
634 }
635 }

```

```

636 \cs_new:Nn \__graph_map_successors_function_aux:Nnn {
637   % #1: function to execute
638   % #2: successor key
639   % #3: successor value
640   #1 {#2} {#3}
641 }
642 \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.

```

643 \cs_new_protected:Nn \graph_map_successors_inline:Nnn {
644   % #1: graph
645   % #2: base vertex
646   % #3: body to execute
647   \withargs (c) [\uniquecsname] [#2] [#3] {
648     \cs_set:Npn ##1 #####2####3 {##3}
649     \graph_map_successors_function:NnN #1 {##2} ##1
650   }
651 }

```

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).

```

652 \cs_new_protected:Nn \graph_map_topological_order_tokens:Nn {
653
654   %% Fill \l__graph_source_vertices with source-nodes and count indegrees
655   %
656   \prop_clear:N \l__graph_source_vertices
657   \graph_map_vertices_inline:Nn #1 {
658     \prop_put:Nnf \l__graph_tmp_indeg_int {##1}
659     { \graph_get_indegree:Nn #1 {##1} }
660     \int_compare:nT {\graph_get_indegree:Nn #1 {##1} = 0} {
661       \prop_put:Nnn \l__graph_source_vertices {##1} {}
662     } }
663
664   %% Main loop
665   %
666   \bool_until_do:nn {\prop_if_empty_p:N \l__graph_source_vertices} {
667     %% Choose any vertex (\l__graph_topo_key_tl, \l__graph_topo_value_tl)
668     %
669     \__graph_prop_any_key_pop:NN
670     \l__graph_source_vertices
671     \l__graph_topo_key_tl
672     \graph_get_vertex:NVNT #1 \l__graph_topo_key_tl \l__graph_topo_val_tl {
673       %% Run the mapping function on the key and value from that vertex
674       %
675       \withargs:VVn \l__graph_topo_key_tl \l__graph_topo_val_tl
676       { #2 {##1} {##2} }
677
678       %% Deduct one from the counter of all affected nodes
679       %% and add all now-empty vertices to \l__graph_source_vertices

```

```

680      %
681      \graph_map_outgoing_edges_inline:NVn #1 \l__graph_topo_key_tl {
682        \prop_put:Nnf \l__graph_tmp_indeg_int {##2}
683        {\int_eval:n {\prop_get:Nn \l__graph_tmp_indeg_int {##2} - 1}}
684        \int_compare:nT {\prop_get:Nn \l__graph_tmp_indeg_int {##2} = 0} {
685          \prop_put:Nnn \l__graph_source_vertices {##2} {}
686        } } } } }
687      \cs_new_protected:Nn \__graph_prop_any_key_pop:NN {
688        \prop_map_inline:Nn #1 {
689          \tl_set:Nn #2 {##1}
690          \prop_remove:Nn #1 {##1}
691          \prop_map_break:n {\use_none:nnn}
692        }
693        \tl_set:Nn #2 {\q_no_value} % TODO: test
694      }
695      \cs_generate_variant:Nn \withargs:nnn {VVn}
696      \cs_generate_variant:Nn \graph_map_outgoing_edges_inline:Nnn {NVn}
697      \cs_generate_variant:Nn \prop_put:Nnn {Nnf}
698      \cs_generate_variant:Nn \graph_get_vertex:NnNT {NVNT}
699      \prop_new:N \l__graph_source_vertices
700      \prop_new:N \l__graph_tmp_indeg_int
701      \tl_new:N \l__graph_topo_key_tl
702      \tl_new:N \l__graph_topo_val_tl

```

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).

```

703 \cs_new:Nn \graph_map_topological_order_function:NN {
704   \graph_map_topological_order_tokens:Nn #1 {#2}
705 }

```

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).

```

706 \cs_new_protected:Nn \graph_map_topological_order_inline:Nn {
707   \withargs (c) [\uniquecsname] [#2] {
708     \cs_set:Npn ##1 #####1####2 {##2}
709     \graph_map_topological_order_function:NN #1 ##1
710   } }

```

3.12 Transforming Graphs

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

```

711 \cs_new_protected:Nn \graph_set_transitive_closure:NN {
712   \__graph_set_transitive_closure:NNNnn #1 #2 \use_none:nnn {} { }
713 }
714 \cs_new_protected:Nn \graph_gset_transitive_closure:NN {
715   \__graph_set_transitive_closure:NNNnn #1 #2 \use_none:nnn {} {g}
716 }
717 \cs_new_protected:Nn \graph_set_transitive_closure:NNNn {
718   \__graph_set_transitive_closure:NNNnn #1 #2 #3 {#4} { }

```

```

719 }
720 \cs_new_protected:Nn \graph_gset_transitive_closure:NNNn {
721   \__graph_set_transitive_closure:NNNnn #1 #2 #3 {#4} {g}
722 }
723 \cs_new_protected:Nn \__graph_set_transitive_closure:NNNnn {
724   % #1: target
725   % #2: source
726   % #3: combination function with argspec :nnn
727   % #4: default 'old' value
728   \use:c{graph_#5set_eq:NN} #1 #2
729
730   \cs_set:Nn \__graph_edge_combinator:nnn {
731     \exp_not:n { #3 {##1} {##2} {##3} } }
732   \cs_generate_variant:Nn \__graph_edge_combinator:nnn {VVV}
733
734   \graph_map_vertices_inline:Nn #2 {
735     \graph_map_vertices_inline:Nn #2 {
736       \graph_get_edge:NnnNT #2 {##1} {####1}
737       \l__graph_edge_value_i_tl {
738         \graph_map_vertices_inline:Nn #2 {
739           \graph_get_edge:NnnNT #2 {####1} {#####1}
740           \l__graph_edge_value_ii_tl {
741             \graph_get_edge:NnnNF #1 {##1} {#####1}
742             \l__graph_edge_value_old_tl {
743               \tl_set:Nn \l__graph_edge_value_old_tl {#4}
744             }
745             \exp_args:NNx \tl_set:No \l__graph_edge_value_new_tl {
746               \__graph_edge_combinator:VVV
747               \l__graph_edge_value_i_tl
748               \l__graph_edge_value_ii_tl
749               \l__graph_edge_value_old_tl
750             }
751             \use:c{graph_#5put_edge:NnnV} #1 {##1} {#####1}
752             \l__graph_edge_value_new_tl
753           } } } } }
754   \cs_generate_variant:Nn \graph_put_edge:Nnnn {NnnV}
755   \cs_generate_variant:Nn \graph_gput_edge:Nnnn {NnnV}
756   \cs_generate_variant:Nn \tl_to_str:n {o}
757   \tl_new:N \l__graph_edge_value_i_tl
758   \tl_new:N \l__graph_edge_value_ii_tl
759   \tl_new:N \l__graph_edge_value_old_tl
760   \tl_new:N \l__graph_edge_value_new_tl

```

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

```

761 \cs_new_protected:Nn \graph_set_transitive_reduction:NN {
762   \__graph_set_transitive_reduction:NNn #1 #2 { } }
763 \cs_new_protected:Nn \graph_gset_transitive_reduction:NN {
764   \__graph_set_transitive_reduction:NNn #1 #2 {g} }
765 \cs_new_protected:Nn \__graph_set_transitive_reduction:NNn {
766   % #1: target
767   % #2: source
768   \use:c{graph_#3set_eq:NN} #1 #2
769   \graph_map_vertices_inline:Nn #2 {

```

```

770 \graph_map_vertices_inline:Nn #2 {
771   \graph_get_edge:NnnNT #2 {##1} {####1} \l_tmpa_tl {
772     \graph_map_vertices_inline:Nn #2 {
773       \graph_get_edge:NnnNT #2 {####1} {#####1} \l_tmpa_tl {
774         \use:c{graph_#3remove_edge:Nnn} #1 {##1} {#####1}
775       } } } } }
776 }

```

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:

```

777 \cs_new_protected:Nn \graph_display_table:N {
778   \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.

```

779 \keys_define:nn {lt3graph-display} {
780   row_keys .bool_set:N = \l__graph_display_row_keys_bool,
781   row_keys .initial:n = {true},
782   row_keys .default:n = {true},
783
784   vertex_vals .bool_set:N = \l__graph_display_vertex_vals_bool,
785   vertex_vals .initial:n = {true},
786   vertex_vals .default:n = {true},
787
788   row_keys_format .tl_set:N = \l__graph_format_row_keys_tl,
789   row_keys_format .initial:n = \textbf,
790   row_keys_format .value_required:,
791
792   col_keys_format .tl_set:N = \l__graph_format_col_keys_tl,
793   col_keys_format .initial:n = \textbf,
794   col_keys_format .value_required:,
795
796   vertex_vals_format .tl_set:N = \l__graph_format_vertex_vals_tl,
797   vertex_vals_format .initial:n = \use:n,
798   vertex_vals_format .value_required:,
799
800   edge_vals_format .tl_set:N = \l__graph_format_edge_vals_tl,
801   edge_vals_format .initial:n = \use:n,
802   edge_vals_format .value_required:,
803
804   edge_diagonal_format .tl_set:N = \l__graph_format_edge_diagonal_tl,
805   edge_diagonal_format .initial:n = \cellcolor{black!30!white},
806   edge_diagonal_format .value_required:,
807
808   edge_direct_format .tl_set:N = \l__graph_format_edge_direct_tl,
809   edge_direct_format .initial:n = \cellcolor{green},
810   edge_direct_format .value_required:,
811

```

```

812 edge_transitive_format .tl_set:N = \l__graph_format_edge_transitive_tl,
813 edge_transitive_format .initial:n = \cellcolor{green!40!yellow}\tiny(tr),
814 edge_transitive_format .value_required:,
815
816 edge_none_format      .tl_set:N = \l__graph_format_edge_none_tl,
817 edge_none_format      .initial:n = {},
818 edge_none_format      .value_required:
819 }

```

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.

```

820 \cs_new_protected:Nn \graph_display_table:Nn {
821   \group_begin:

```

We process those options passed with #2:

```

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

```

We populate the top row of the table:

```

823   \tl_put_right:Nn \l__graph_table_content_tl {\hline}
824   \seq_clear:N \l__graph_row_seq
825   \bool_if:NT \l__graph_display_row_keys_bool
826     { \seq_put_right:Nn \l__graph_row_seq {
827       \tl_put_right:Nn \l__graph_table_colspec_tl {|r|} }
828   \bool_if:NT \l__graph_display_vertex_vals_bool
829     { \seq_put_right:Nn \l__graph_row_seq {
830       \tl_put_right:Nn \l__graph_table_colspec_tl {|c|} }
831   \graph_map_vertices_inline:Nn #1 {
832     \tl_put_right:Nn \l__graph_table_colspec_tl {|c}
833     \seq_put_right:Nn \l__graph_row_seq
834       { { \l__graph_format_col_keys_tl {##1} } }
835   }
836   \tl_put_right:Nn \l__graph_table_colspec_tl {|}
837   \tl_put_right:Nx \l__graph_table_content_tl
838     { \seq_use:Nn \l__graph_row_seq {&} }
839   \tl_put_right:Nn \l__graph_table_content_tl
840     { \\hline\hline }

```

We populate the remaining rows:

```

841   \graph_map_vertices_inline:Nn #1 {
842     \seq_clear:N \l__graph_row_seq
843     \bool_if:NT \l__graph_display_row_keys_bool {
844       \seq_put_right:Nn \l__graph_row_seq
845         { { \l__graph_format_row_keys_tl {##1} } } }
846     \bool_if:NT \l__graph_display_vertex_vals_bool {
847       \seq_put_right:Nn \l__graph_row_seq
848         { { \l__graph_format_vertex_vals_tl {##2} } } }
849     \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:

```

850 \graph_get_edge:NnnNTF #1 {##1} {####1} \l_tmpa_tl {
851 \quark_if_no_value:VF \l_tmpa_tl {
852 \tl_set_eq:NN \l__graph_cell_content_tl \l_tmpa_tl
853 \tl_set:Nf \l__graph_cell_content_tl
854 { \exp_args:NV \l__graph_format_edge_direct_tl
855 \l__graph_cell_content_tl } }
856 }{\graph_acyclic_if_path_exist:NnnTF #1 {##1} {####1} {
857 \tl_set_eq:NN \l__graph_cell_content_tl
858 \l__graph_format_edge_transitive_tl
859 }{
860 \tl_set_eq:NN \l__graph_cell_content_tl
861 \l__graph_format_edge_none_tl
862 }}

```

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

```

863 \str_if_eq:nnT {##1} {####1} {
864 \tl_set:Nf \l__graph_cell_content_tl
865 { \exp_args:NV \l__graph_format_edge_diagonal_tl
866 \l__graph_cell_content_tl } }

```

Tertiary formatting is applied to all vertex value cells:

```

867 \tl_set:Nf \l__graph_cell_content_tl
868 { \exp_args:NV \l__graph_format_edge_vals_tl
869 \l__graph_cell_content_tl }

```

We can now add the cell to the row sequence:

```

870 \seq_put_right:NV \l__graph_row_seq \l__graph_cell_content_tl
871 }

```

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

```

872 \tl_put_right:Nx \l__graph_table_content_tl
873 { \seq_use:Nn \l__graph_row_seq {&} }
874 \tl_put_right:Nn \l__graph_table_content_tl {\hline}
875 }

```

Finally, we print the table itself:

```

876 \withargs:VVn \l__graph_table_colspec_tl \l__graph_table_content_tl
877 { \begin{tabular}{##1}##2\end{tabular} }
878 \group_end:
879 }

```

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

```

880 \cs_generate_variant:Nn \quark_if_no_value:nF {VF}
881 \cs_generate_variant:Nn \withargs:nnn {VVn}
882 \tl_new:N \l__graph_table_colspec_tl
883 \tl_new:N \l__graph_table_content_tl
884 \tl_new:N \l__graph_cell_content_tl
885 \bool_new:N \l__graph_table_skipfirst_bool
886 \seq_new:N \l__graph_row_seq

```

Change History

0.0.1	0.1.0
General: initial version 1	General: fixed a bug in \graph_(g)put_vertex and \graph_(g)put_edge, which caused their global versions not to work properly 1
0.0.8	
General: a great many untracked changes 1	
0.0.9	
General: creation of the documentation 1	

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	
\\ 840, 874	_graph_if_vertex_can_reach_cycle:NnnTF 311
_graph_acyclic_if_path_exist:Nnnnn 394, 398	_graph_if_vertex_can_reach_cycle:Nnnnn 324, 328
_graph_clear:Nn 68, 70, 71	_graph_if_vertex_can_reach_cycle_p:Nnn 309, 343
_graph_clear_new:Nn 76, 78, 79	_graph_if_vertex_can_reach_cycle_p:Nno 338
_graph_edge_combinator:VVV 746	_graph_key:n 31
_graph_edge_combinator:nnn 730, 732	_graph_key:nn 32, 161, 165, 170, 174, 242, 245, 248, 251, 285, 304
_graph_empty_set 21, 309, 311, 313, 315	_graph_key:nnn 33
_graph_for_each_prop_datatype:n 36, 62, 72, 90	_graph_key:nnnn 34
_graph_get_cycle:NN 373	_graph_key:nnnnn 35
_graph_get_cycle:NNTF 368	_graph_map_edges_function_aux:Nnn 454, 456
_graph_gput_edges_from:NNn 260, 262, 263	_graph_map_edges_function_aux:Nnnn 458, 460
_graph_if_cyclic:Nnn 349, 353	_graph_map_edges_function_aux:Nnnv 457
_graph_if_vertex_can_reach_cycle:Nnn 317	_graph_map_edges_tokens_aux:nnn 444, 446
_graph_if_vertex_can_reach_cycle:NnnF 315	
_graph_if_vertex_can_reach_cycle:NnnT 313	

__graph_map_edges_tokens_aux:nnnn	__graph_remove_edge:Nnnn	219, 221, 222
..... 448, 450	__graph_remove_vertex:Nnn	193, 195, 196
__graph_map_edges_tokens_aux:nnnv	__graph_set_cons:nn	23, 339
__graph_map_incoming_edges_function_aux:nnnn	__graph_set_eq:NNn	85, 87, 88
..... 495, 497	__graph_set_if_in:nn	13
__graph_map_incoming_edges_function_aux:nnnn	__graph_set_if_in_p:nn	337
..... 503, 510	__graph_set_transitive_closure:NNNnn	
__graph_map_incoming_edges_function_aux:nnnv	__graph_set_transitive_reduction:NNn	
..... 502		
__graph_map_incoming_edges_tokens_aux:nnnn		762, 764, 765
..... 473, 475	__graph_tl:n	26
__graph_map_incoming_edges_tokens_aux:nnnn	__graph_tl:nn	27
..... 481, 488	__graph_tl:nnn	28,
__graph_map_incoming_edges_tokens_aux:nnnv	63, 73, 93, 94, 117, 123, 124, 143,	
..... 480	145, 150, 152, 160, 164, 169, 173,	
__graph_map_outgoing_edges_function_aux:nnnn	211, 212, 213, 226, 228, 233, 235,	
..... 548, 550	241, 244, 247, 250, 276, 284, 294,	
__graph_map_outgoing_edges_function_aux:nnnn	303, 413, 416, 424, 432, 443, 453,	
..... 556, 563	472, 494, 525, 547, 578, 598, 613, 633	
__graph_map_outgoing_edges_function_aux:nnnn	__graph_tl:nnnn	29
..... 555	__graph_tl:nnnnn	30
__graph_map_outgoing_edges_tokens_aux:nnnn	__prg_break_point:	18
..... 526, 528	__prop_if_in:nwn	15
__graph_map_outgoing_edges_tokens_aux:nnnn	__prop_pair:wn	16, 24
..... 534, 541		
__graph_map_outgoing_edges_tokens_aux:nnnv	B	
..... 533	\begin	877
__graph_map_successors_function_aux:NnNnn	\bool_if:NT	825, 828, 843, 846
..... 614, 616	\bool_if:nT	335, 358, 405
__graph_map_successors_function_aux:NnNnn	\bool_new:N	885
..... 622, 624	\bool_until_do:nn	666
__graph_map_successors_function_aux:Nnn		
..... 636, 642	C	
__graph_map_successors_function_aux:Nnv	\cellcolor	805, 809, 813
..... 632	\cs_generate_variant:Nn	
__graph_map_successors_tokens_aux:NNnnn	66, 186, 187, 188, 216, 254, 255,	
..... 579, 581	256, 257, 343, 429, 450, 460, 488,	
__graph_map_successors_tokens_aux:NNnnnn	510, 541, 563, 607, 642, 695, 696,	
..... 587, 589	697, 698, 732, 754, 755, 756, 880, 881	
__graph_map_successors_tokens_aux:nnn	\cs_if_exist:NF	99
..... 601, 607	\cs_if_exist:Np	97
__graph_map_successors_tokens_aux:nnv	\cs_if_exist:NT	98
..... 597	\cs_if_exist:NTF	100
__graph_map_vertices_tokens_aux:nnn	\cs_new:Nn	23,
..... 427, 429	26, 27, 28, 29, 30, 31, 32, 33, 34, 35,	
__graph_map_vertices_tokens_aux:nnv	328, 353, 398, 412, 415, 418, 422,	
..... 425	427, 430, 441, 446, 448, 451, 456,	
__graph_prop_any_key_pop:NN	458, 467, 475, 481, 489, 497, 503,	
__graph_ptr_gnew:Nn	520, 528, 534, 542, 550, 556, 573,	
__graph_ptr_new:Nn	581, 589, 601, 608, 616, 624, 636, 703	
__graph_put_edge:Nnnnn	\cs_new:Npn	308, 310, 312, 314
..... 129, 131, 133, 135, 136	\cs_new_protected:Nn	36, 42, 49, 56, 67,
__graph_put_vertex:Nnnn	69, 71, 75, 77, 79, 84, 86, 88, 101,	
..... 102, 104, 106, 108, 109	103, 105, 107, 109, 128, 130, 132,	

134, 136, 192, 194, 196, 218, 220, 222, 259, 261, 263, 435, 461, 511, 564, 643, 652, 687, 706, 711, 714, 717, 720, 723, 761, 763, 765, 777, 820	\cs_set:Nn 730	\graph_if_exist:NT 98
\cs_set:Npn . . . 437, 463, 516, 569, 648, 708	\cs_set_eq:NN 21, 97, 98, 99, 100	\graph_if_exist:NTF 57, 100
E		\graph_if_exists:NF 80
\end 877		\graph_if_vertex_can_reach_cycle:NnF 314
\exp_args:Nnv 433		\graph_if_vertex_can_reach_cycle:NnT 312
\exp_args:NNx 745		\graph_if_vertex_can_reach_cycle:NnTF 310
\exp_args:NV 854, 865, 868		\graph_if_vertex_can_reach_cycle_p:Nn 308, 359
\exp_not:n 731		\graph_if_vertex_exist:Nn 290
G		\graph_if_vertex_exist:NnT 266, 267
\g__graph_prop_data_types_seq 37, 38, 39		\graph_map_edges_function:NN . . . 451, 464
\graph_acyclic_if_path_exist:Nnn . . 387		\graph_map_edges_inline:Nn 265, 461
\graph_acyclic_if_path_exist:NnnTF 856		\graph_map_edges_tokens:Nn 441
\graph_acyclic_if_path_exist_p:Nnn 408		\graph_map_incoming_edges_function:NnN 489, 517
\graph_clear:N 67		\graph_map_incoming_edges_inline:Nnn 206, 511
\graph_clear_new:N 75		\graph_map_incoming_edges_tokens:Nnn 467
\graph_display_table:N 777		\graph_map_outgoing_edges_function:NnN 542, 570
\graph_display_table:Nn 778, 820		\graph_map_outgoing_edges_inline:Nnn 201, 564, 696
\graph_gclear:N 69		\graph_map_outgoing_edges_inline:Nvn 681
\graph_gclear_new:N 77		\graph_map_outgoing_edges_tokens:Nnn 323, 393, 520
\graph_get_cycle:NN 362		\graph_map_successors_function:NnN 608, 649
\graph_get_degree:Nn 418		\graph_map_successors_inline:Nnn 378, 643
\graph_get_edge:NnnN 280		\graph_map_successors_tokens:Nnn . . . 573
\graph_get_edge:NnnNF 140, 741		\graph_map_topological_order_function:NN 703, 709
\graph_get_edge:NnnNT 223, 736, 739, 771, 773		\graph_map_topological_order_inline:Nn 706
\graph_get_edge:NnnNTF 850		\graph_map_topological_order_tokens:Nn 652, 704
\graph_get_indegree:Nn 415, 420, 659, 660		\graph_map_vertices_function:NN 430, 438
\graph_get_outdegree:Nn 412, 419		\graph_map_vertices_inline:Nn 435, 657, 734, 735, 738, 769, 770, 772, 831, 841, 849
\graph_get_vertex:NnN 273		\graph_map_vertices_tokens:Nn 348, 422
\graph_get_vertex:NnNT . . . 120, 198, 698		\graph_new:N 56, 81
\graph_get_vertex:NnNTF 138, 139		\graph_put_edge:Nnn 128
\graph_get_vertex:NvNT 672		\graph_put_edge:Nnnn 132, 754
\graph_gput_edge:Nnn 130		\graph_put_edges_from:NN 259
\graph_gput_edge:Nnnn 134, 268, 755		\graph_put_vertex:Nn 101
\graph_gput_edges_from:NN 261		\graph_put_vertex:Nnn 105
\graph_gput_vertex:Nn 103		\graph_remove_edge:Nnn 218
\graph_gput_vertex:Nnn 107		\graph_remove_vertex:Nn 192
\graph_gremove_edge:Nnn 220		
\graph_gremove_vertex:Nn 194		
\graph_gset_eq:NN 86		
\graph_gset_transitive_closure:NN . . 714		
\graph_gset_transitive_closure:NNNn 720		
\graph_gset_transitive_reduction:NN 763		
\graph_if_cyclic:N 344		
\graph_if_edge_exist:Nnn 299		
\graph_if_exist:NF 99		
\graph_if_exist:Np 97		

