

# The `lt3graph` package<sup>\*†</sup>

Michiel Helvensteijn  
[mhelvens+latex@gmail.com](mailto:mhelvens+latex@gmail.com)

December 7, 2013

---

Development of this package is organized at [github.com/mhelvens/latex-lt3graph](https://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.<sup>1</sup> 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:

---

<sup>\*</sup>This document corresponds to `lt3graph` v0.0.9-r1, dated 2013/12/05.

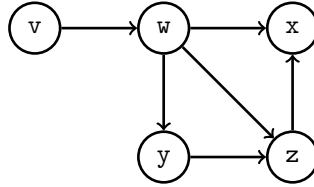
<sup>†</sup>The prefix `lt3` indicates that this package is a user-contributed `expl3` library, in contrast to packages prefixed with `l3`, which are officially supported by the  $\text{\LaTeX}3$  team.

<sup>1</sup> 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.<sup>2</sup>

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:

<sup>2</sup>This 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-r1}
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}
```

We now test whether the `display` option was used:

```
13 \bool_new:N \__graph_format_bool
14 \bool_set_false:N \__graph_format_bool
15
16 \keys_define:nn {lt3graph}
17 { display .code:n = \bool_set_true:N \__graph_format_bool }
18
19 \ProcessKeysPackageOptions {lt3graph}
```

If the `display` option was used, load additional packages:

```
20 \bool_if:NT \__graph_format_bool {
21   \RequirePackage[table]{xcolor}
22 }
```

### 3.3 Additions to L<sup>A</sup>T<sub>E</sub>X3 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.

```
23 \prg_new_conditional:Npnn \__graph_set_if_in:nn #1#2 { p }
24 {
25   \__prop_if_in:nwn {#2} #1 \s_obj_end
26   \__prop_pair:wn #2 \s__prop { }
27   \q_recursion_tail
28   \__prg_break_point:
29 }
30
31 \cs_set_eq:NN \__graph_empty_set \s__prop
```

```

32
33 \cs_new:Nn \__graph_set_cons:nn {
34 #1 \__prop_pair:wn #2 \s__prop {}
35 }

```

### 3.4 Data Access

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

```

36 \cs_new:Nn \__graph_tl:n      { g__graph_data (#1)          _tl }
37 \cs_new:Nn \__graph_tl:nn     { g__graph_data (#1) (#2)      _tl }
38 \cs_new:Nn \__graph_tl:nnn    { g__graph_data (#1) (#2) (#3)  _tl }
39 \cs_new:Nn \__graph_tl:nnnn   { g__graph_data (#1) (#2) (#3) (#4) _tl }
40 \cs_new:Nn \__graph_tl:nnnnn  { g__graph_data (#1) (#2) (#3) (#4) (#5) _tl }

```

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

```

41 \cs_new:Nn \__graph_key:n      { key (#1)                    }
42 \cs_new:Nn \__graph_key:nn     { key (#1) (#2)                }
43 \cs_new:Nn \__graph_key:nnn    { key (#1) (#2) (#3)           }
44 \cs_new:Nn \__graph_key:nnnn   { key (#1) (#2) (#3) (#4)      }
45 \cs_new:Nn \__graph_key:nnnnn  { key (#1) (#2) (#3) (#4) (#5) }

```

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

```

46 \cs_new_protected:Nn \__graph_for_each_prop_datatype:n
47 { \seq_map_inline:Nn \g__graph_prop_data_types_seq {#1} }
48 \seq_new:N          \g__graph_prop_data_types_seq
49 \seq_set_from_clist:Nn \g__graph_prop_data_types_seq
50 {vertices, edge-values, edge-froms, edge-tos, edge-triples,
51 indegree, outdegree}

```

### 3.5 Storing data through pointers

The following function embodies a L<sup>A</sup>T<sub>E</sub>X3 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.

```

52 \cs_new_protected:Nn \__graph_ptr_new:Nn {
53   \withargs [\uniquecsname] {
54     \tl_set:Nn #1 {##1}
55     \tl_new:c   {##1}
56     \tl_set:cn  {##1} {#2}
57   }
58 }

```

### 3.6 Creating and initializing graphs

Globally create a new graph:

```

59 \cs_new_protected:Nn \graph_new:N {
60   \graph_if_exist:NTF #1 {
61     % TODO: error
62   }{
63     \tl_new:N #1
64     \tl_set:Nf #1 { \tl_trim_spaces:f {\str_tail:n{#1}} }
65     \__graph_for_each_prop_datatype:n
66       { \prop_new:c {\__graph_tl:nnn{graph}{#1}{##1}} }
67   }
68 }
69 \cs_generate_variant:Nn \tl_trim_spaces:n {f}

```

Remove all data from a graph:

```

70 \cs_new_protected:Nn \graph_clear:N
71   { \__graph_clear:Nn #1 { } }
72 \cs_new_protected:Nn \graph_gclear:N
73   { \__graph_clear:Nn #1 {g} }
74 \cs_new_protected:Nn \__graph_clear:Nn {
75   \__graph_for_each_prop_datatype:n
76     { \use:c{prop_#2clear:c} {\__graph_tl:nnn{graph}{#1}{##1}} }
77 }

```

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

```

78 \cs_new_protected:Nn \graph_clear_new:N
79   { \__graph_clear_new:Nn #1 { } }
80 \cs_new_protected:Nn \graph_gclear_new:N
81   { \__graph_clear_new:Nn #1 {g} }
82 \cs_new_protected:Nn \__graph_clear_new:Nn {
83   \graph_if_exists:NF #1
84   { \graph_new:N #1 }
85   \use:c{graph_#2clear:N} #1
86 }

```

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

```

87 \cs_new_protected:Nn \graph_set_eq:NN
88   { \__graph_set_eq:NNn #1 #2 { } }
89 \cs_new_protected:Nn \graph_gset_eq:NN
90   { \__graph_set_eq:NNn #1 #2 {g} }
91 \cs_new_protected:Nn \__graph_set_eq:NNn {
92   \use:c{graph_#3clear:N} #1
93   \__graph_for_each_prop_datatype:n
94     {
95       \use:c{prop_#3set_eq:cc}
96       {\__graph_tl:nnn{graph}{#1}{##1}}
97       {\__graph_tl:nnn{graph}{#2}{##1}}
98     }
99 }

```

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)`:

```

100 \cs_set_eq:NN \graph_if_exist:Np \cs_if_exist:Np
101 \cs_set_eq:NN \graph_if_exist:NT \cs_if_exist:NT
102 \cs_set_eq:NN \graph_if_exist:NF \cs_if_exist:NF
103 \cs_set_eq:NN \graph_if_exist:NTF \cs_if_exist:NTF

```

### 3.7 Manipulating graphs

Put a new vertex inside a graph:

```

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

```

Put a new edge inside a graph:

```

131 \cs_new_protected:Nn \graph_put_edge:Nnn
132 { \__graph_put_edge:Nnnnn #1 {#2} {#3} {} { } }
133 \cs_new_protected:Nn \graph_gput_edge:Nnn
134 { \__graph_put_edge:Nnnnn #1 {#2} {#3} {} {g} }
135 \cs_new_protected:Nn \graph_put_edge:Nnnn
136 { \__graph_put_edge:Nnnnn #1 {#2} {#3} {#4} { } }
137 \cs_new_protected:Nn \graph_gput_edge:Nnnn
138 { \__graph_put_edge:Nnnnn #1 {#2} {#3} {#4} {g} }
139 \cs_new_protected:Nn \__graph_put_edge:Nnnnn
140 {
141   \graph_get_vertex:NnNTF #1 {#2} \l__graph_from_value_tl {
142     \graph_get_vertex:NnNTF #1 {#3} \l__graph_to_value_tl {
143       \graph_get_edge:NnnNF #1 {#2} {#3} \l_tmpa_tl {
144         %% increment outgoing degree of vertex #2
145         %

```



```

146     \use:c{prop_#5put:cnf} {\_graph_tl:nnn{graph}{#1}{outdegree}} {#2}
147     {\int_eval:n {
148         \prop_get:cn {\_graph_tl:nnn{graph}{#1}{outdegree}} {#2} + 1
149     }}
150
151     %%% increment incoming degree of vertex #3
152     %
153     \use:c{prop_#5put:cnf} {\_graph_tl:nnn{graph}{#1}{indegree}} {#3}
154     {\int_eval:n {
155         \prop_get:cn {\_graph_tl:nnn{graph}{#1}{indegree}} {#3} + 1
156     }}
157 }
158
159 %%% actually add the edge
160 %
161 \withargs:VVn \l__graph_from_value_tl \l__graph_to_value_tl {
162     \use:c{prop_#5put:cox}
163     { \_graph_tl:nnn{graph}{#1}{edge-froms} }
164     { \_graph_key:nn{#2}{#3} }
165     { \tl_to_str:n{#2} }
166     \use:c{prop_#5put:cox}
167     { \_graph_tl:nnn{graph}{#1}{edge-tos} }
168     { \_graph_key:nn{#2}{#3} }
169     { \tl_to_str:n{#3} }
170     \_graph_ptr_new:Nn \l__graph_edge_data_tl {#4}
171     \use:c{prop_#5put:coV}
172     { \_graph_tl:nnn{graph}{#1}{edge-values} }
173     { \_graph_key:nn{#2}{#3} }
174     \l__graph_edge_data_tl
175     \use:c{prop_#5put:cox}
176     { \_graph_tl:nnn{graph}{#1}{edge-triples} }
177     { \_graph_key:nn{#2}{#3} }
178     { {\tl_to_str:n{#2}} }
179     { {\tl_to_str:n{#3}} }
180     { \l__graph_edge_data_tl }
181 }
182 }{
183     % TODO: Error ('to' vertex doesn't exist)
184 }
185 }{
186     % TODO: Error ('from' vertex doesn't exist)
187 }
188 }
189 \cs_generate_variant:Nn \prop_gput:Nnn {cox, coV, cnf}
190 \cs_generate_variant:Nn \prop_put:Nnn {cox, coV, cnf}
191 \cs_generate_variant:Nn \withargs:nnn {VVn}
192 \tl_new:N \l__graph_edge_data_tl
193 \tl_new:N \l__graph_from_value_tl
194 \tl_new:N \l__graph_to_value_tl

```

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

```

195 \cs_new_protected:Nn \graph_remove_vertex:Nn
196 { \_graph_remove_vertex:Nnn #1 {#2} { } }

```

```

197 \cs_new_protected:Nn \graph_gremove_vertex:Nn
198 { \__graph_remove_vertex:Nnn #1 {#2} {g} }
199 \cs_new_protected:Nn \__graph_remove_vertex:Nnn
200 {
201   \graph_get_vertex:NnNT #1 {#2} \l__graph_vertex_data_tl {
202     %%% remove outgoing edges
203     %
204     \graph_map_outgoing_edges_inline:Nnn #1 {#2}
205     { \use:c{graph_#3remove_edge:Nnn} #1 {##1} {##2} }
206
207     %%% remove incoming edges
208     %
209     \graph_map_incoming_edges_inline:Nnn #1 {#2}
210     { \use:c{graph_#3remove_edge:Nnn} #1 {##1} {##2} }
211
212     %%% remove the vertex
213     %
214     \use{prop_#3remove:cn} {\__graph_tl:nnn{graph}{#1}{vertices}} {#2}
215     \use{prop_#3remove:cn} {\__graph_tl:nnn{graph}{#1}{indegree}} {#2}
216     \use{prop_#3remove:cn} {\__graph_tl:nnn{graph}{#1}{outdegree}} {#2}
217   }
218 }
219 \cs_generate_variant:Nn \prop_put:Nnn {cnV}
220 % \tl_new:N \l__graph_vertex_data_tl % reusing from other function

```

Remove an edge from the graph:

```

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

```

```

248     { \_graph_key:nn{#2}{#3} }
249     \use:c{prop_#4remove:co}
250     { \_graph_tl:nnn{graph}{#1}{edge-values} }
251     { \_graph_key:nn{#2}{#3} }
252     \use:c{prop_#4remove:co}
253     { \_graph_tl:nnn{graph}{#1}{edge-triples} }
254     { \_graph_key:nn{#2}{#3} }
255 }
256 }
257 \cs_generate_variant:Nn \prop_remove:Nn {co}
258 \cs_generate_variant:Nn \prop_gremove:Nn {co}
259 \cs_generate_variant:Nn \prop_put:Nnn {cnf}
260 \cs_generate_variant:Nn \prop_gput:Nnn {cnf}
261 %\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:

```

262 \cs_new_protected:Nn \graph_put_edges_from:NN
263 { \_graph_gput_edges_from:NNn #1 #2 { } }
264 \cs_new_protected:Nn \graph_gput_edges_from:NN
265 { \_graph_gput_edges_from:NNn #1 #2 {g} }
266 \cs_new_protected:Nn \_graph_gput_edges_from:NNn
267 {
268   \graph_map_edges_inline:Nn #2 {
269     \graph_if_vertex_exist:NnT #1 {##1} {
270       \graph_if_vertex_exist:NnT #1 {##2} {
271         \graph_gput_edge:Nnnn #1 {##1} {##2} {##3}
272       }
273     }
274   }
275 }

```

### 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.

```

276 \prg_new_protected_conditional:Nnn \graph_get_vertex:NnN
277 {T, F, TF}
278 {
279   \prop_get:cnNTF { \_graph_tl:nnn {graph} {#1} {vertices} } {#2} #3
280   { \tl_set:Nv #3 {#3} \prg_return_true: }
281   { \prg_return_false: }
282 }

```

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.

```

283 \prg_new_protected_conditional:Nnn \graph_get_edge:NnnN
284 {T, F, TF}
285 {
286   \prop_get:cnNTF
287   { \_graph_tl:nnn{graph}{#1}{edge-values} }
288   { \_graph_key:nn{#2}{#3} }

```

```

289     #4
290     { \tl_set:Nv #4 {#4} \prg_return_true: }
291     { \prg_return_false: }
292 }

```

### 3.9 Graph Conditionals

An expandable test for the existence of a vertex:

```

293 \prg_new_conditional:Nnn \graph_if_vertex_exist:Nn
294 {p, T, F, TF}
295 {
296   \prop_if_in:cnTF
297   { \__graph_tl:nnn {graph} {#1} {vertices} }
298   { #2 }
299   { \prg_return_true: }
300   { \prg_return_false: }
301 }

```

An expandable test for the existence of an edge:

```

302 \prg_new_conditional:Nnn \graph_if_edge_exist:Nnn
303 {p, T, F, TF}
304 {
305   \prop_if_in:coTF
306   { \__graph_tl:nnn {graph} {#1} {edge-values} }
307   { \__graph_key:nn{#2}{#3} }
308   { \prg_return_true: }
309   { \prg_return_false: }
310 }

```

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

```

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

```

```

333 % #2: visited vertices in 'prop literal' format (internal l3prop)
334 % #3: start vertex (not used)
335 % #4: current vertex
336 % #5: edge value (behind ptr, not used)
337 {
338   \bool_if:nT
339   {
340     \__graph_set_if_in_p:nn {#2} {#4} ||
341     \__graph_if_vertex_can_reach_cycle_p:Nno #1 {#4}
342     { \__graph_set_cons:nn {#2} {#4} }
343   }
344   { \prop_map_break:n {\use_i:nn \prg_return_true:} }
345 }
346 \cs_generate_variant:Nn \__graph_if_vertex_can_reach_cycle_p:Nnn {Nno}

```

Test whether graph #1 contains any cycles:

```

347 \prg_new_conditional:Nnn \graph_if_cyclic:N
348 {p, T, F, TF}
349 % #1: graph id
350 {
351   \graph_map_vertices_tokens:Nn #1
352   { \__graph_if_cyclic:Nnn #1 }
353   \prg_return_false:
354 }
355
356 \cs_new:Nn \__graph_if_cyclic:Nnn
357 % #1: graph id
358 % #2: vertex id
359 % #3: vertex value (not used)
360 {
361   \bool_if:nT
362   { \graph_if_vertex_can_reach_cycle_p:Nn #1 {#2} }
363   { \prop_map_break:n {\use_i:nn \prg_return_true:} }
364 }

```

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

```

365 \prg_new_conditional:Nnn \graph_acyclic_if_path_exist:Nnn
366 {p, T, F, TF}
367 % #1: graph id
368 % #2: start vertex
369 % #3: end vertex
370 {
371   \graph_map_outgoing_edges_tokens:Nnn #1 {#2}
372   { \__graph_acyclic_if_path_exist:Nnnnn #1 {#3} }
373   \prg_return_false:
374 }
375
376 \cs_new:Nn \__graph_acyclic_if_path_exist:Nnnnn
377 % #1: graph id
378 % #2: end vertex
379 % #3: start vertex (not used)
380 % #4: possible end vertex

```

```

381 % #5: edge value (behind ptr, do not use)
382 {
383     \bool_if:nT
384     {
385         \str_if_eq_p:nn {#4} {#2} ||
386         \graph_acyclic_if_path_exist_p:Nnn #1 {#4} {#2}
387     }
388     { \prop_map_break:n {\use_i:nn \prg_return_true:} }
389 }

```

### 3.10 Querying Information

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

```

390 \cs_new:Nn \graph_get_outdegree:Nn {
391     \prop_get:cn {\__graph_tl:nnn{graph}{#1}{outdegree}} {#2}
392 }

```

Get the number of edges leading into vertex #2:

```

393 \cs_new:Nn \graph_get_indegree:Nn {
394     \prop_get:cn {\__graph_tl:nnn{graph}{#1}{indegree}} {#2}
395 }

```

Get the number of edges connected to vertex #2:

```

396 \cs_new:Nn \graph_get_degree:Nn {
397     \int_eval:n{ \graph_get_outdegree:Nn #1 {#2} +
398                 \graph_get_indegree:Nn #1 {#2} }
399 }

```

### 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.

```

400 \cs_new:Nn \graph_map_vertices_tokens:Nn {
401     \prop_map_tokens:cn
402     { \__graph_tl:nnn{graph}{#1}{vertices} }
403     { \__graph_map_vertices_tokens_aux:nnv {#2} }
404 }
405 \cs_new:Nn \__graph_map_vertices_tokens_aux:nnn
406     { #1 {#2} {#3} }
407 \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.

```

408 \cs_new:Nn \graph_map_vertices_function:NN {
409     \prop_map_tokens:cn
410     { \__graph_tl:nnn{graph}{#1}{vertices} }
411     { \exp_args:Nnv #2 }
412 }

```

---

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.

```

413 \cs_new_protected:Nn \graph_map_vertices_inline:Nn {
414   \withargs (c) [\uniquecsname] [#2] {
415     \cs_set:Npn ##1 #####1####2 {##2}
416     \graph_map_vertices_function:NN #1 ##1
417   }
418 }

```

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.

```

419 \cs_new:Nn \graph_map_edges_tokens:Nn {
420   \prop_map_tokens:cn
421   { \__graph_tl:nnn{graph}{#1}{edge-triples} }
422   { \__graph_map_edges_tokens_aux:nnn {#2} }
423 }
424 \cs_new:Nn \__graph_map_edges_tokens_aux:nnn
425   { \__graph_map_edges_tokens_aux:nnnv {#1} #3 }
426 \cs_new:Nn \__graph_map_edges_tokens_aux:nnnn
427   { #1 {#2} {#3} {#4} }
428 \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.

```

429 \cs_new:Nn \graph_map_edges_function:NN {
430   \prop_map_tokens:cn
431   { \__graph_tl:nnn{graph}{#1}{edge-triples} }
432   { \__graph_map_edges_function_aux:Nnn #2 }
433 }
434 \cs_new:Nn \__graph_map_edges_function_aux:Nnn
435   { \__graph_map_edges_function_aux:Nnnv #1 #3 }
436 \cs_new:Nn \__graph_map_edges_function_aux:Nnnn
437   { #1 {#2} {#3} {#4} }
438 \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.

```

439 \cs_new_protected:Nn \graph_map_edges_inline:Nn {
440   \withargs (c) [\uniquecsname] [#2] {
441     \cs_set:Npn ##1 #####1####2####3 {##2}
442     \graph_map_edges_function:NN #1 ##1
443   }
444 }

```

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.

```

445 \cs_new:Nn \graph_map_incoming_edges_tokens:Nnn {
446   % #1: graph
447   % #2: base vertex
448   % #3: tokens to execute
449   \prop_map_tokens:cn
450   { \__graph_tl:nnn{graph}{#1}{edge-triples} }
451   { \__graph_map_incoming_edges_tokens_aux:nnnn {#2} {#3} }
452 }
453 \cs_new:Nn \__graph_map_incoming_edges_tokens_aux:nnnn
454   % #1: base vertex
455   % #2: tokens to execute
456   % #3: edge key
457   % #4: edge-triple {from}{to}{value}
458   { \__graph_map_incoming_edges_tokens_aux:nnnnv {#1} {#2} #4 }
459 \cs_new:Nn \__graph_map_incoming_edges_tokens_aux:nnnnn
460   % #1: base vertex
461   % #2: tokens to execute
462   % #3: edge 'from' vertex
463   % #4: edge 'to' vertex
464   % #5: edge value
465   { \str_if_eq:nnT {#1} {#4} { #2 {#3} {#4} {#5} } }
466 \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.

```

467 \cs_new:Nn \graph_map_incoming_edges_function:NnN {
468   % #1: graph
469   % #2: base vertex
470   % #3: function to execute
471   \prop_map_tokens:cn
472   { \__graph_tl:nnn{graph}{#1}{edge-triples} }
473   { \__graph_map_incoming_edges_function_aux:nNnn {#2} #3 }
474 }
475 \cs_new:Nn \__graph_map_incoming_edges_function_aux:nNnn
476   % #1: base vertex
477   % #2: function to execute
478   % #3: edge key
479   % #4: edge-triple {from}{to}{value}
480   { \__graph_map_incoming_edges_function_aux:nNnnv {#1} #2 #4 }
481 \cs_new:Nn \__graph_map_incoming_edges_function_aux:nNnnn
482   % #1: base vertex
483   % #2: function 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_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.



```

489 \cs_new_protected:Nn \graph_map_incoming_edges_inline:Nnn {
490   % #1: graph
491   % #2: base vertex
492   % #3: body to execute
493   \withargs (c) [\uniquecsname] [#2] [#3] {
494     \cs_set:Npn ##1 #####2####3 {##3}
495     \graph_map_incoming_edges_function:NnN #1 {##2} ##1
496   }
497 }

```

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.

```

498 \cs_new:Nn \graph_map_outgoing_edges_tokens:Nnn {
499   % #1: graph
500   % #2: base vertex
501   % #3: tokens to execute
502   \prop_map_tokens:cn
503     { \__graph_tl:nnn{graph}{#1}{edge-triples} }
504     { \__graph_map_outgoing_edges_tokens_aux:nnnn {#2} {#3} }
505 }
506 \cs_new:Nn \__graph_map_outgoing_edges_tokens_aux:nnnn
507   % #1: base vertex
508   % #2: tokens to execute
509   % #3: edge key (not used)
510   % #4: edge-triple {from}{to}{value}
511   { \__graph_map_outgoing_edges_tokens_aux:nnnnv {#1} {#2} #4 }
512 \cs_new:Nn \__graph_map_outgoing_edges_tokens_aux:nnnnn
513   % #1: base vertex
514   % #2: tokens to execute
515   % #3: edge ‘from’ vertex
516   % #4: edge ‘to’ vertex
517   % #5: edge value
518   { \str_if_eq:nnT {#1} {#3} { #2 {#3} {#4} {#5} } }
519 \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.

```

520 \cs_new:Nn \graph_map_outgoing_edges_function:NnN {
521   % #1: graph
522   % #2: base vertex
523   % #3: function to execute
524   \prop_map_tokens:cn
525     { \__graph_tl:nnn{graph}{#1}{edge-triples} }
526     { \__graph_map_outgoing_edges_function_aux:nNnn {#2} #3 }
527 }
528 \cs_new:Nn \__graph_map_outgoing_edges_function_aux:nNnn
529   % #1: base vertex
530   % #2: function to execute
531   % #3: edge key
532   % #4: edge-triple {from}{to}{value}
533   { \__graph_map_outgoing_edges_function_aux:nNnnv {#1} #2 #4 }

```

```

534 \cs_new:Nn \__graph_map_outgoing_edges_function_aux:nNnnn
535 % #1: base vertex
536 % #2: function 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_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.

```

542 \cs_new_protected:Nn \graph_map_outgoing_edges_inline:Nnn {
543 % #1: graph
544 % #2: base vertex
545 % #3: body to execute
546 \withargs (c) [\uniquecsname] [#2] [#3] {
547   \cs_set:Npn ##1 #####2####3 {##3}
548   \graph_map_outgoing_edges_function:NnN #1 {##2} ##1
549 }
550 }

```

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.

```

551 \cs_new:Nn \graph_map_successors_tokens:Nnn {
552 % #1: graph
553 % #2: base vertex
554 % #3: tokens to execute
555 \prop_map_tokens:cn
556 { \__graph_tl:nnn{graph}{#1}{edge-triples} }
557 { \__graph_map_successors_tokens_aux:Nnnnn #1 {#2} {#3} }
558 }
559 \cs_new:Nn \__graph_map_successors_tokens_aux:Nnnnn {
560 % #1: the graph
561 % #2: base vertex
562 % #3: tokens to execute
563 % #4: edge key (not used)
564 % #5: edge-triple {from}{to}{value}
565 \__graph_map_successors_tokens_aux:Nnnnnn #1 {#2} {#3} #5
566 }
567 \cs_new:Nn \__graph_map_successors_tokens_aux:Nnnnnn {
568 % #1: the graph
569 % #2: base vertex
570 % #3: tokens to execute
571 % #4: edge 'from' vertex
572 % #5: edge 'to' vertex
573 % #6: ptr to edge value (not used)
574 \str_if_eq:nnT {#2} {#4} {
575   \__graph_map_successors_tokens_aux:nnv
576   {#3} {#5} {\prop_get:cn{\__graph_tl:nnn{graph}{#1}{vertices}}{#5}}
577 }
578 }

```

```

579 \cs_new:Nn \__graph_map_successors_tokens_aux:nnn {
580   % #1: tokens to execute
581   % #2: successor key
582   % #3: successor value
583   #1 {#2} {#3}
584 }
585 \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.

```

586 \cs_new:Nn \graph_map_successors_function:NnN {
587   % #1: graph
588   % #2: base vertex
589   % #3: function to execute
590   \prop_map_tokens:cn
591     { \__graph_tl:nnn{graph}{#1}{edge-triples} }
592     { \__graph_map_successors_function_aux:NnNnn #1 {#2} #3 }
593 }
594 \cs_new:Nn \__graph_map_successors_function_aux:NnNnn {
595   % #1: the graph
596   % #2: base vertex
597   % #3: function to execute
598   % #4: edge key (not used)
599   % #5: edge-triple {from}{to}{value}
600   \__graph_map_successors_function_aux:NnNnnn #1 {#2} #3 #5
601 }
602 \cs_new:Nn \__graph_map_successors_function_aux:NnNnnn {
603   % #1: the graph
604   % #2: base vertex
605   % #3: function to execute
606   % #4: edge 'from' vertex
607   % #5: edge 'to' vertex
608   % #6: ptr to edge value (not used)
609   \str_if_eq:nnT {#2} {#4} {
610     \__graph_map_successors_function_aux:Nnv
611       #3 {#5} {\prop_get:cn{\__graph_tl:nnn{graph}{#1}{vertices}}{#5}}
612   }
613 }
614 \cs_new:Nn \__graph_map_successors_function_aux:Nnn {
615   % #1: function to execute
616   % #2: successor key
617   % #3: successor value
618   #1 {#2} {#3}
619 }
620 \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.

```

621 \cs_new_protected:Nn \graph_map_successors_inline:Nnn {
622   % #1: graph
623   % #2: base vertex

```

```

624 % #3: body to execute
625 \withargs (c) [\uniquecsname] [#2] [#3] {
626   \cs_set:Npn ##1 #####2####3 {##3}
627   \graph_map_successors_function:NnN #1 {##2} ##1
628 }
629 }

```

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).

```

630 \cs_new_protected:Nn \graph_map_topological_order_tokens:Nn {
631   %% Fill \l__graph_source_vertices with source-nodes and count indegrees
632   %
633   \prop_clear:N \l__graph_source_vertices
634   \graph_map_vertices_inline:Nn #1 {
635     \prop_put:Nnf \l__graph_tmp_indegreess_int {##1}
636     {\graph_get_indegree:Nn #1 {##1}}
637     \int_compare:nT {\graph_get_indegree:Nn #1 {##1} = 0}
638     { \prop_put:Nnn \l__graph_source_vertices {##1} {} }
639   }
640
641   %% Main loop
642   %
643   \bool_until_do:nn {\prop_if_empty_p:N \l__graph_source_vertices} {
644     %% Choose any vertex (\l__graph_topo_key_tl, \l__graph_topo_value_tl)
645     %
646     \__graph_prop_any_key_pop:NN \l__graph_source_vertices \l__graph_topo_key_tl
647     \graph_get_vertex:NVNT #1 \l__graph_topo_key_tl \l__graph_topo_val_tl {
648       %% Run the mapping funtion on the key and value from that vertex
649       %
650       \withargs:VVn \l__graph_topo_key_tl \l__graph_topo_val_tl
651       { #2 {##1} {##2} }
652
653       %% Deduct one from the counter of all affected nodes
654       %% and add all now-empty vertices to \l__graph_source_vertices
655       %
656       \graph_map_outgoing_edges_inline:NVn #1 \l__graph_topo_key_tl {
657         \prop_put:Nnf \l__graph_tmp_indegreess_int {##2}
658         {\int_eval:n {\prop_get:Nn \l__graph_tmp_indegreess_int {##2} - 1}}
659         \int_compare:nT {\prop_get:Nn \l__graph_tmp_indegreess_int {##2} = 0} {
660           \prop_put:Nnn \l__graph_source_vertices {##2} {}
661         }
662       }
663     }
664   }
665 }
666 \cs_new_protected:Nn \__graph_prop_any_key_pop:NN {
667   \prop_map_inline:Nn #1 {
668     \tl_set:Nn #2 {##1}
669     \prop_remove:Nn #1 {##1}
670     \prop_map_break:n {\use_none:n}
671   }
672   { \tl_set:Nn #2 {\q_no_value} }

```

```

673 }
674 \cs_generate_variant:Nn \withargs:nnn {VVn}
675 \cs_generate_variant:Nn \graph_map_outgoing_edges_inline:Nnn {NVn}
676 \cs_generate_variant:Nn \prop_put:Nnn {Nnf}
677 \cs_generate_variant:Nn \graph_get_vertex:NnNT {NVNT}
678 \prop_new:N \l__graph_source_vertices
679 \prop_new:N \l__graph_tmp_indegreess_int
680 \tl_new:N \l__graph_topo_key_tl
681 \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).

```

682 \cs_new:Nn \graph_map_topological_order_function:NN
683 { \graph_map_topological_order_tokens:Nn #1 {#2} }

```

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).

```

684 \cs_new_protected:Nn \graph_map_topological_order_inline:Nn {
685   \withargs (c) [\uniquecsname] [#2] {
686     \cs_set:Npn ##1 #####1####2 {##2}
687     \graph_map_topological_order_function:NN #1 ##1
688   }
689 }

```

### 3.12 Transforming Graphs

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

```

690 \cs_new_protected:Nn \graph_set_transitive_closure:NN
691 { \__graph_set_transitive_closure:NNNnn #1 #2 \use_none:nnn {} { } }
692 \cs_new_protected:Nn \graph_gset_transitive_closure:NN
693 { \__graph_set_transitive_closure:NNNnn #1 #2 \use_none:nnn {} {g} }
694 \cs_new_protected:Nn \graph_set_transitive_closure:NNNn
695 { \__graph_set_transitive_closure:NNNnn #1 #2 #3 {#4} { } }
696 \cs_new_protected:Nn \graph_gset_transitive_closure:NNNn
697 { \__graph_set_transitive_closure:NNNnn #1 #2 #3 {#4} {g} }
698 \cs_new_protected:Nn \__graph_set_transitive_closure:NNNnn
699 % #1: target
700 % #2: source
701 % #3: combination function with argspec :nnn
702 % #4: default 'old' value
703 {
704   \use:c{graph_#5set_eq:NN} #1 #2
705
706   \cs_set:Nn \__graph_edge_combinator:nnn
707   { \exp_not:n { #3 {##1} {##2} {##3} } }
708   \cs_generate_variant:Nn \__graph_edge_combinator:nnn {VVV}
709
710   \graph_map_vertices_inline:Nn #2 {
711     \graph_map_vertices_inline:Nn #2 {

```

```

712 \graph_get_edge:NnnNT #2 {##1} {####1} \l__graph_edge_value_i_tl {
713 \graph_map_vertices_inline:Nn #2 {
714 \graph_get_edge:NnnNT #2 {####1} {#####1}
715 \l__graph_edge_value_ii_tl {
716 \graph_get_edge:NnnNF #1 {##1} {#####1}
717 \l__graph_edge_value_old_tl
718 { \tl_set:Nn \l__graph_edge_value_old_tl {#4} }
719 \exp_args:NNx \tl_set:No \l__graph_edge_value_new_tl {
720 \__graph_edge_combinator:VVV
721 \l__graph_edge_value_i_tl
722 \l__graph_edge_value_ii_tl
723 \l__graph_edge_value_old_tl
724 }
725 \use:c{graph_#5put_edge:NnnV} #1 {##1} {#####1}
726 \l__graph_edge_value_new_tl
727 }
728 }
729 }
730 }
731 }
732 }
733 \cs_generate_variant:Nn \graph_put_edge:Nnnn {NnnV}
734 \cs_generate_variant:Nn \graph_gput_edge:Nnnn {NnnV}
735 \cs_generate_variant:Nn \tl_to_str:n {o}
736 \tl_new:N \l__graph_edge_value_i_tl
737 \tl_new:N \l__graph_edge_value_ii_tl
738 \tl_new:N \l__graph_edge_value_old_tl
739 \tl_new:N \l__graph_edge_value_new_tl

```

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

```

740 \cs_new_protected:Nn \graph_set_transitive_reduction:NN
741 { \__graph_set_transitive_reduction:NNn #1 #2 { } }
742 \cs_new_protected:Nn \graph_gset_transitive_reduction:NN
743 { \__graph_set_transitive_reduction:NNn #1 #2 {g} }
744 \cs_new_protected:Nn \__graph_set_transitive_reduction:NNn
745 % #1: target
746 % #2: source
747 {
748 \use:c{graph_#3set_eq:NN} #1 #2
749 \graph_map_vertices_inline:Nn #2 {
750 \graph_map_vertices_inline:Nn #2 {
751 \graph_get_edge:NnnNT #2 {##1} {####1} \l_tmpa_tl {
752 \graph_map_vertices_inline:Nn #2 {
753 \graph_get_edge:NnnNT #2 {####1} {#####1} \l_tmpa_tl {
754 \use:c{graph_#3remove_edge:Nnn} #1 {##1} {#####1}
755 }
756 }
757 }
758 }
759 }
760 }

```

### 3.13 Displaying Graphs

If the `display` option was given, we define some additional functions that can display the graph in table-form.

```
761 \bool_if:NT \__graph_format_bool {
```

This is the option-less version, which delegates to the full version of the function:

```
762 \cs_new_protected:Nn \graph_display_table:N
763 { \graph_display_table:Nn #1 {} }
```

The full version has a second argument accepting options that determine table formatting. We first define those options:

```
764 \keys_define:nn {lt3graph-display} {
765   row_keys .bool_set:N = \l__graph_display_row_keys_bool,
766   row_keys .initial:n = {true},
767   row_keys .default:n = {true},
768
769   vertex_vals .bool_set:N = \l__graph_display_vertex_vals_bool,
770   vertex_vals .initial:n = {true},
771   vertex_vals .default:n = {true},
772
773   row_keys_format .tl_set:N = \l__graph_format_row_keys_tl,
774   row_keys_format .initial:n = \textbf,
775   row_keys_format .value_required:,
776
777   col_keys_format .tl_set:N = \l__graph_format_col_keys_tl,
778   col_keys_format .initial:n = \textbf,
779   col_keys_format .value_required:,
780
781   vertex_vals_format .tl_set:N = \l__graph_format_vertex_vals_tl,
782   vertex_vals_format .initial:n = \use:n,
783   vertex_vals_format .value_required:,
784
785   edge_vals_format .tl_set:N = \l__graph_format_edge_vals_tl,
786   edge_vals_format .initial:n = \use:n,
787   edge_vals_format .value_required:,
788
789   edge_diagonal_format .tl_set:N = \l__graph_format_edge_diagonal_tl,
790   edge_diagonal_format .initial:n = \cellcolor{black!30!white},
791   edge_diagonal_format .value_required:,
792
793   edge_direct_format .tl_set:N = \l__graph_format_edge_direct_tl,
794   edge_direct_format .initial:n = \cellcolor{green},
795   edge_direct_format .value_required:,
796
797   edge_transitive_format .tl_set:N = \l__graph_format_edge_transitive_tl,
798   edge_transitive_format .initial:n = \cellcolor{green!40!yellow}\tiny(tr),
799   edge_transitive_format .value_required:,
800
801   edge_none_format .tl_set:N = \l__graph_format_edge_none_tl,
802   edge_none_format .initial:n = {},
803   edge_none_format .value_required:
```

```
804 }
```

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.

```
805 \cs_new_protected:Nn \graph_display_table:Nn {
806   \group_begin:
```

We process those options passed with #2:

```
807   \keys_set:nn {lt3graph-display} {#2}
```

We populate the top row of the table:

```
808   \tl_put_right:Nn \l__graph_table_content_tl {\hline}
809   \seq_clear:N \l__graph_row_seq
810   \bool_if:NT \l__graph_display_row_keys_bool
811     { \seq_put_right:Nn \l__graph_row_seq {
812       \tl_put_right:Nn \l__graph_table_colspec_tl {|r|} }
813   \bool_if:NT \l__graph_display_vertex_vals_bool
814     { \seq_put_right:Nn \l__graph_row_seq {
815       \tl_put_right:Nn \l__graph_table_colspec_tl {|c|} }
816   \graph_map_vertices_inline:Nn #1 {
817     \tl_put_right:Nn \l__graph_table_colspec_tl {|c}
818     \seq_put_right:Nn \l__graph_row_seq
819       { { \l__graph_format_col_keys_tl {##1} } }
820   }
821   \tl_put_right:Nn \l__graph_table_colspec_tl {|}|
822   \tl_put_right:Nx \l__graph_table_content_tl
823     { \seq_use:Nn \l__graph_row_seq {&} }
824   \tl_put_right:Nn \l__graph_table_content_tl
825     { \\hline\hline }
```

We populate the remaining rows:

```
826   \graph_map_vertices_inline:Nn #1 {
827     \seq_clear:N \l__graph_row_seq
828     \bool_if:NT \l__graph_display_row_keys_bool {
829       \seq_put_right:Nn \l__graph_row_seq
830         { { \l__graph_format_row_keys_tl {##1} } } }
831     \bool_if:NT \l__graph_display_vertex_vals_bool {
832       \seq_put_right:Nn \l__graph_row_seq
833         { { \l__graph_format_vertex_vals_tl {##2} } } }
834     \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:

```
835     \graph_get_edge:NnnNTF #1 {##1} {####1} \l_tmpa_tl {
836       \quark_if_no_value:VF \l_tmpa_tl {
837         \tl_set_eq:NN \l__graph_cell_content_tl \l_tmpa_tl
838         \tl_set:Nf \l__graph_cell_content_tl
839           { \exp_args:NV \l__graph_format_edge_direct_tl
840             \l__graph_cell_content_tl } }
```



```

841     }\graph_acyclic_if_path_exist:NnnTF #1 {##1} {####1} {
842         \tl_set_eq:NN \l__graph_cell_content_tl
843         \l__graph_format_edge_transitive_tl
844     }{
845         \tl_set_eq:NN \l__graph_cell_content_tl
846         \l__graph_format_edge_none_tl
847     }}

```

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

```

848     \str_if_eq:nnT {##1} {####1} {
849         \tl_set:Nf \l__graph_cell_content_tl
850         { \exp_args:NV \l__graph_format_edge_diagonal_tl
851           \l__graph_cell_content_tl } }

```

Tertiary formatting is applied to all vertex value cells:

```

852     \tl_set:Nf \l__graph_cell_content_tl
853     { \exp_args:NV \l__graph_format_edge_vals_tl
854       \l__graph_cell_content_tl }

```

We can now add the cell to the row sequence:

```

855     \seq_put_right:NV \l__graph_row_seq \l__graph_cell_content_tl
856 }

```

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

```

857     \tl_put_right:Nx \l__graph_table_content_tl { \seq_use:Nn \l__graph_row_seq {&} }
858     \tl_put_right:Nn \l__graph_table_content_tl {\hline}
859 }

```

Finally, we print the table itself:

```

860     \withargs:VVn \l__graph_table_colspec_tl \l__graph_table_content_tl
861     { \begin{tabular}{##1}##2\end{tabular} }
862     \group_end:
863 }

```

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

```

864 \cs_generate_variant:Nn \quark_if_no_value:nF {VF}
865 \cs_generate_variant:Nn \withargs:nnn {VVn}
866 \tl_new:N \l__graph_table_colspec_tl
867 \tl_new:N \l__graph_table_content_tl
868 \tl_new:N \l__graph_cell_content_tl
869 \bool_new:N \l__graph_table_skipfirst_bool
870 \seq_new:N \l__graph_row_seq

```

## Change History

0.0.1	changes	1
General: initial version	1	
0.0.8	0.0.9	
General: a great many untracked	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	
\\	825, 858
\\_graph_acyclic_if_path_exist:Nnnnn	372, 376
\\_graph_clear:Nn	71, 73, 74
\\_graph_clear_new:Nn	79, 81, 82
\\_graph_edge_combinator:VVV	720
\\_graph_edge_combinator:nnn	706, 708
\\_graph_empty_set	31, 312, 314, 316, 318
\\_graph_for_each_prop_datatype:n	46, 65, 75, 93
\\_graph_format_bool	13, 14, 17, 20, 761
\\_graph_gput_edges_from:NNn	263, 265, 266
\\_graph_if_cyclic:Nnn	352, 356
\\_graph_if_vertex_can_reach_cycle:Nnn	320
\\_graph_if_vertex_can_reach_cycle:NnnF	318
\\_graph_if_vertex_can_reach_cycle:NnnT	316
\\_graph_if_vertex_can_reach_cycle:NnnTF	314
\\_graph_if_vertex_can_reach_cycle:Nnnnn	327, 331
\\_graph_if_vertex_can_reach_cycle_p:Nnn	312, 346
\\_graph_if_vertex_can_reach_cycle_p:Nno	341
\\_graph_key:n	41
\\_graph_key:nn	42, 164, 168, 173, 177, 245, 248, 251, 254, 288, 307
\\_graph_key:nnn	43
\\_graph_key:nnnn	44
\\_graph_key:nnnnn	45
\\_graph_map_edges_function_aux:Nnn	432, 434
\\_graph_map_edges_function_aux:Nnnn	436, 438
\\_graph_map_edges_function_aux:Nnnv	435
\\_graph_map_edges_tokens_aux:nnn	422, 424
\\_graph_map_edges_tokens_aux:nnnn	426, 428
\\_graph_map_edges_tokens_aux:nnnv	425
\\_graph_map_incoming_edges_function_aux:nNnn	473, 475
\\_graph_map_incoming_edges_function_aux:nNnnn	481, 488
\\_graph_map_incoming_edges_function_aux:nNnnv	480
\\_graph_map_incoming_edges_tokens_aux:nnnn	451, 453
\\_graph_map_incoming_edges_tokens_aux:nnnnn	459, 466
\\_graph_map_incoming_edges_tokens_aux:nnnnv	458
\\_graph_map_outgoing_edges_function_aux:nNnn	526, 528



G			
\g__graph_prop_data_types_seq	47, 48, 49	\graph_map_incoming_edges_function:NnN	467, 495
\graph_acyclic_if_path_exist:Nnn	.. 365	\graph_map_incoming_edges_inline:Nnn	209, 489
\graph_acyclic_if_path_exist:NnnTF	841	\graph_map_incoming_edges_tokens:Nnn	445
\graph_acyclic_if_path_exist_p:Nnn	386	\graph_map_outgoing_edges_function:NnN	520, 548
\graph_clear:N	70	\graph_map_outgoing_edges_inline:Nnn	204, 542, 675
\graph_clear_new:N	78	\graph_map_outgoing_edges_inline:NvN	656
\graph_display_table:N	762	\graph_map_outgoing_edges_tokens:Nnn	326, 371, 498
\graph_display_table:Nn	763, 805	\graph_map_successors_function:NnN	586, 627
\graph_gclear:N	72	\graph_map_successors_inline:Nnn	.. 621
\graph_gclear_new:N	80	\graph_map_successors_tokens:Nnn	.. 551
\graph_get_degree:Nn	396	\graph_map_topological_order_function:NN	682, 687
\graph_get_edge:NnnN	283	\graph_map_topological_order_inline:Nn	684
\graph_get_edge:NnnNF	143, 716	\graph_map_topological_order_tokens:Nn	630, 683
\graph_get_edge:NnnNT	226, 712, 714, 751, 753	\graph_map_vertices_function:NN	408, 416
\graph_get_edge:NnnNTF	835	\graph_map_vertices_inline:Nn	413, 634, 710, 711, 713, 749, 750, 752, 816, 826, 834
\graph_get_indegree:Nn	393, 398, 636, 637	\graph_map_vertices_tokens:Nn	.. 351, 400
\graph_get_outdegree:Nn	390, 397	\graph_new:N	59, 84
\graph_get_vertex:NnN	276	\graph_put_edge:Nnn	131
\graph_get_vertex:NnNT	123, 201, 677	\graph_put_edge:Nnnn	135, 733
\graph_get_vertex:NnNTF	141, 142	\graph_put_edges_from:NN	262
\graph_get_vertex:NVNT	647	\graph_put_vertex:Nn	104
\graph_gput_edge:Nnn	133	\graph_put_vertex:Nnn	108
\graph_gput_edge:Nnnn	137, 271, 734	\graph_remove_edge:Nnn	221
\graph_gput_edges_from:NN	264	\graph_remove_vertex:Nn	195
\graph_gput_vertex:Nn	106	\graph_set_eq:NN	87
\graph_gput_vertex:Nnn	110	\graph_set_transitive_closure:NN	.. 690
\graph_gremove_edge:Nnn	223	\graph_set_transitive_closure:NNNn	694
\graph_gremove_vertex:Nn	197	\graph_set_transitive_reduction:NN	740
\graph_gset_eq:NN	89	\group_begin:	806
\graph_gset_transitive_closure:NN	692	\group_end:	862
\graph_gset_transitive_closure:NNNn	696	H	
\graph_gset_transitive_reduction:NN	742	\hline	808, 825, 858
\graph_if_cyclic:N	347	I	
\graph_if_edge_exist:Nnn	302	\int_compare:nT	637, 659
\graph_if_exist:Nf	102	\int_eval:n	147, 154, 230, 237, 397, 658
\graph_if_exist:Np	100	K	
\graph_if_exist:NT	101	\keys_define:nn	16, 764
\graph_if_exist:NTF	60, 103	\keys_set:nn	807
\graph_if_exists:Nf	83		
\graph_if_vertex_can_reach_cycle:NnF	317		
\graph_if_vertex_can_reach_cycle:NnT	315		
\graph_if_vertex_can_reach_cycle:NnTF	313		
\graph_if_vertex_can_reach_cycle_p:Nn	311, 362		
\graph_if_vertex_exist:Nn	293		
\graph_if_vertex_exist:NnT	269, 270		
\graph_map_edges_function:NN	429, 442		
\graph_map_edges_inline:Nn	268, 439		
\graph_map_edges_tokens:Nn	419		

<b>L</b>	
\l__graph_cell_content_tl .....	
..... 837, 838, 840,	
842, 845, 849, 851, 852, 854, 855, 868	
\l__graph_display_row_keys_bool .....	
..... 765, 810, 828	
\l__graph_display_vertex_vals_bool ..	
..... 769, 813, 831	
\l__graph_edge_data_tl .....	
..... 170, 174, 180, 192, 226, 261	
\l__graph_edge_value_i_tl .. 712, 721, 736	
\l__graph_edge_value_ii_tl 715, 722, 737	
\l__graph_edge_value_new_tl 719, 726, 739	
\l__graph_edge_value_old_tl .....	
..... 717, 718, 723, 738	
\l__graph_format_col_keys_tl .. 777, 819	
\l__graph_format_edge_diagonal_tl ..	
..... 789, 850	
\l__graph_format_edge_direct_tl 793, 839	
\l__graph_format_edge_none_tl .. 801, 846	
\l__graph_format_edge_transitive_tl	
..... 797, 843	
\l__graph_format_edge_vals_tl .. 785, 853	
\l__graph_format_row_keys_tl .. 773, 830	
\l__graph_format_vertex_vals_tl 781, 833	
\l__graph_from_value_tl .. 141, 161, 193	
\l__graph_row_seq .... 809, 811, 814,	
818, 823, 827, 829, 832, 855, 857, 870	
\l__graph_source_vertices .....	
631, 633, 638, 643, 646, 654, 660, 678	
\l__graph_table_colspec_tl .....	
..... 812, 815, 817, 821, 860, 866	
\l__graph_table_content_tl .....	
.... 808, 822, 824, 857, 858, 860, 867	
\l__graph_table_skipfirst_bool ... 869	
\l__graph_tmp_indegrees_int .....	
..... 635, 657, 658, 659, 679	
\l__graph_to_value_tl .... 142, 161, 194	
\l__graph_topo_key_tl .....	
..... 644, 646, 647, 650, 656, 680	
\l__graph_topo_val_tl .... 647, 650, 681	
\l__graph_topo_value_tl .....	
644	
\l__graph_vertex_data_tl .....	
..... 116, 121, 130, 201, 220	
\l_tmpa_tl 123, 143, 751, 753, 835, 836, 837	
<b>N</b>	
\NeedsTeXFormat .....	1
<b>P</b>	
\prg_new_conditional:Nnn .....	
..... 293, 302, 320, 347, 365	
\prg_new_conditional:Npnn .....	23
\prg_new_protected_conditional:Nnn ..	
..... 276, 283	
\prg_return_false: .....	
.... 281, 291, 300, 309, 328, 353, 373	
\prg_return_true: .....	
.... 280, 290, 299, 308, 344, 363, 388	
\ProcessKeysPackageOptions .....	19
\prop_clear:N .....	633
\prop_get:cn .....	
148, 155, 231, 238, 391, 394, 576, 611	
\prop_get:cnNTF .....	279
\prop_get:coNTF .....	286
\prop_get:Nn .....	658, 659
\prop_gput:Nnn .....	189, 260
\prop_gremove:Nn .....	258
\prop_if_empty_p:N .....	643
\prop_if_in:cnTF .....	296
\prop_if_in:coTF .....	305
\prop_map_break:n .... 344, 363, 388, 670	
\prop_map_inline:Nn .....	667
\prop_map_tokens:cn .....	401, 409,
420, 430, 449, 471, 502, 524, 555, 590	
\prop_new:c .....	66
\prop_new:N .....	678, 679
\prop_put:Nnf .....	635, 657
\prop_put:Nnn .. 190, 219, 259, 638, 660, 676	
\prop_remove:Nn .....	257, 669
\ProvidesExplPackage .....	3
<b>Q</b>	
\q_no_value .....	672
\q_recursion_tail .....	27
\quark_if_no_value:nF .....	864
\quark_if_no_value:VF .....	836
<b>R</b>	
\RequirePackage 2, 5, 6, 7, 8, 9, 10, 11, 12, 21	
<b>S</b>	
\s__prop .....	26, 31, 34
\s_obj_end .....	25
\seq_clear:N .....	809, 827
\seq_map_inline:Nn .....	47
\seq_new:N .....	48, 870
\seq_put_right:Nn 811, 814, 818, 829, 832	
\seq_put_right:NV .....	855
\seq_set_from_clist:Nn .....	49
\seq_use:Nn .....	823, 857
\str_if_eq:nnT .....	
.... 465, 487, 518, 540, 574, 609, 848	
\str_if_eq_p:nn .....	385
\str_tail:n .....	64
<b>T</b>	
\textbf 774, 778	

<code>\tiny</code> . . . . .	798				<b>U</b>
<code>\tl_new:c</code> . . . . .	55	<code>\uniquecsname</code>	53, 414, 440, 493, 546, 625, 685		
<code>\tl_new:N</code> . . . . .	63,	<code>\use</code> . . . . .	214, 215, 216		
	130, 192, 193, 194, 220, 261, 680,	<code>\use:c</code> . . . . .	76, 85,		
	681, 736, 737, 738, 739, 866, 867, 868		92, 95, 120, 126, 127, 146, 153, 162,		
<code>\tl_put_right:Nn</code> . . . . .			166, 171, 175, 205, 210, 229, 236,		
	808, 812, 815, 817, 821, 824, 858		243, 246, 249, 252, 704, 725, 748, 754		
<code>\tl_put_right:Nx</code> . . . . .	822, 857	<code>\use:n</code> . . . . .	782, 786		
<code>\tl_set:cn</code> . . . . .	56	<code>\use_i:nn</code> . . . . .	344, 363, 388		
<code>\tl_set:Nf</code> . . . . .	64, 838, 849, 852	<code>\use_none:n</code> . . . . .	670		
<code>\tl_set:Nn</code> . . . . .	54, 668, 672, 718	<code>\use_none:nnn</code> . . . . .	691, 693		
<code>\tl_set:No</code> . . . . .	719				
<code>\tl_set:Nv</code> . . . . .	280, 290				
<code>\tl_set_eq:NN</code> . . . . .	837, 842, 845				<b>W</b>
<code>\tl_to_str:n</code> . . . . .	165, 169, 178, 179, 735	<code>\withargs</code> .	53, 414, 440, 493, 546, 625, 685		
<code>\tl_trim_spaces:f</code> . . . . .	64	<code>\withargs:nnn</code> . . . . .	191, 674, 865		
<code>\tl_trim_spaces:n</code> . . . . .	69	<code>\withargs:VVn</code> . . . . .	161, 650, 860		