The lt3graph package*†

Michiel Helvensteijn
mhelvens+latex@gmail.com

December 7, 2013

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

---

1 Introduction

This package provides a data-structure for use in the LaTeX3 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:

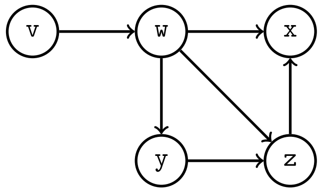
```
\ExplSyntaxOn
    \graph_new:N          \l_my_graph
    \graph_put_vertex:Nn \l_my_graph {v}
    \graph_put_vertex:Nn \l_my_graph {w}
    \graph_put_vertex:Nn \l_my_graph {x}
    \graph_put_vertex:Nn \l_my_graph {y}
    \graph_put_vertex:Nn \l_my_graph {z}
    \graph_put_edge:Nnn  \l_my_graph {v} {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-r1, 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 LaTeX3 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:

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

```
\NeedsTeXFormat{LaTeX2e}
\RequirePackage{expl3}
\ProvidesExplPackage{lt3graph}{2013/12/05}{0.0.9-r1}
  {a LaTeX3 datastructure for representing directed graphs with data}
```

3.2 Required Packages

These are the packages we'll need:

```
\RequirePackage{l3candidates}
\RequirePackage{l3clist}
\RequirePackage{l3keys2e}
\RequirePackage{l3msg}
\RequirePackage{l3prg}
\RequirePackage{l3prop}
\RequirePackage{xparse}
\RequirePackage{withargs}
```

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

```
\bool_new:N       \__graph_format_bool
\bool_set_false:N \__graph_format_bool

\keys_define:nn {lt3graph}
  { display .code:n = \bool_set_true:N \__graph_format_bool }

\ProcessKeysPackageOptions {lt3graph}
```

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

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

3.3 Additions to LaTeX3 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.

```
\prg_new_conditional:Npnn \__graph_set_if_in:nn #1#2 { p }
  {
    \__prop_if_in:nwwn {#2} #1 \s_obj_end
      \__prop_pair:wn #2 \s__prop { }
      \q_recursion_tail
    \__prg_break_point:
  }

\cs_set_eq:NN \__graph_empty_set \s__prop
```

```

\cs_new:Nn \__graph_set_cons:nn {
#1 \__prop_pair:wn #2 \s__prop {}
}
```

3.4 Data Access

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

```
\cs_new:Nn \__graph_tl:n      { g__graph_data (#1)                     _tl }
\cs_new:Nn \__graph_tl:nn     { g__graph_data (#1) (#2)                _tl }
\cs_new:Nn \__graph_tl:nnn    { g__graph_data (#1) (#2) (#3)           _tl }
\cs_new:Nn \__graph_tl:nnnn   { g__graph_data (#1) (#2) (#3) (#4)      _tl }
\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:

```
\cs_new:Nn \__graph_key:n     { key (#1)                     }
\cs_new:Nn \__graph_key:nn    { key (#1) (#2)                }
\cs_new:Nn \__graph_key:nnn   { key (#1) (#2) (#3)           }
\cs_new:Nn \__graph_key:nnnn  { key (#1) (#2) (#3) (#4)      }
\cs_new:Nn \__graph_key:nnnnn { key (#1) (#2) (#3) (#4) (#5) }
```

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

```
\cs_new_protected:Nn \__graph_for_each_prop_datatype:n
  { \seq_map_inline:Nn \g__graph_prop_data_types_seq {#1} }
\seq_new:N               \g__graph_prop_data_types_seq
\seq_set_from_clist:Nn   \g__graph_prop_data_types_seq
  {vertices, edge-values, edge-froms, edge-tos, edge-triples,
   indegree, outdegree}
```

3.5 Storing data through pointers

The following function embodies a LaTeX3 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.

```
\cs_new_protected:Nn \__graph_ptr_new:Nn {
  \withargs [\uniquecsname] {
    \tl_set:Nn #1 {##1}
    \tl_new:c     {##1}
    \tl_set:cn    {##1} {#2}
  }
}
```

3.6 Creating and initializing graphs

Globally create a new graph:

```
\cs_new_protected:Nn \graph_new:N {
  \graph_if_exist:NTF #1 {
    % TODO: error
  }{
    \tl_new:N #1
    \tl_set:Nf #1 { \tl_trim_spaces:f {\str_tail:n{#1}} }
    \__graph_for_each_prop_datatype:n
      { \prop_new:c {\__graph_tl:nnn{graph}{#1}{##1}} }
  }
}
\cs_generate_variant:Nn \tl_trim_spaces:n {f}
```

Remove all data from a graph:

```
\cs_new_protected:Nn \graph_clear:N
  {\__graph_clear:Nn #1 { } }
\cs_new_protected:Nn \graph_gclear:N
  {\__graph_clear:Nn #1 {g} }
\cs_new_protected:Nn \__graph_clear:Nn {
  \__graph_for_each_prop_datatype:n
    { \use:c{prop_#2clear:c} {\__graph_tl:nnn{graph}{#1}{##1}} }
}
```

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

```
\cs_new_protected:Nn \graph_clear_new:N
  { \__graph_clear_new:Nn #1 { } }
\cs_new_protected:Nn \graph_gclear_new:N
  { \__graph_clear_new:Nn #1 {g} }
\cs_new_protected:Nn \__graph_clear_new:Nn {
  \graph_if_exists:NF #1
    { \graph_new:N #1 }
  \use:c{graph_#2clear:N} #1
}
```

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

```
\cs_new_protected:Nn \graph_set_eq:NN
  { \__graph_set_eq:NNn #1 #2 { } }
\cs_new_protected:Nn \graph_gset_eq:NN
  { \__graph_set_eq:NNn #1 #2 {g} }
\cs_new_protected:Nn \__graph_set_eq:NNn {
  \use:c{graph_#3clear:N} #1
  \__graph_for_each_prop_datatype:n
    {
      \use:c{prop_#3set_eq:cc}
        {\__graph_tl:nnn{graph}{#1}{##1}}
        {\__graph_tl:nnn{graph}{#2}{##1}}
    }
}
```

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)`:

```
\cs_set_eq:NN \graph_if_exist:Np  \cs_if_exist:Np
\cs_set_eq:NN \graph_if_exist:NT  \cs_if_exist:NT
\cs_set_eq:NN \graph_if_exist:NF  \cs_if_exist:NF
\cs_set_eq:NN \graph_if_exist:NTF \cs_if_exist:NTF
```

3.7 Manipulating graphs

Put a new vertex inside a graph:

```
\cs_new_protected:Nn \graph_put_vertex:Nn
  { \__graph_put_vertex:Nnnn #1 {#2} {} { } }
\cs_new_protected:Nn \graph_gput_vertex:Nn
  { \__graph_put_vertex:Nnnn #1 {#2} {} {g} }
\cs_new_protected:Nn \graph_put_vertex:Nnn
  { \__graph_put_vertex:Nnnn #1 {#2} {#3} { } }
\cs_new_protected:Nn \graph_gput_vertex:Nnn
  { \__graph_put_vertex:Nnnn #1 {#2} {#3} {g} }
\cs_new_protected:Nn \__graph_put_vertex:Nnnn
  {
    %%% create pointer to value
    %
    \__graph_ptr_new:Nn \l__graph_vertex_data_tl {#3}

    %%% add the vertex
    %
    \use:c {prop_#4put:cnV} {\__graph_tl:nnn{graph}{#1}{vertices}}
        {#2} \l__graph_vertex_data_tl

    \graph_get_vertex:NnNT #1 {#2} \l_tmpa_tl {
      %%% initialize degree to 0
      %
      \use:c{prop_#4put:cnn} {\__graph_tl:nnn{graph}{#1}{indegree}}  {#2}{0}
      \use:c{prop_#4put:cnn} {\__graph_tl:nnn{graph}{#1}{outdegree}} {#2}{0}
    }
  }
\tl_new:N \l__graph_vertex_data_tl
```

Put a new edge inside a graph:

```
\cs_new_protected:Nn \graph_put_edge:Nnn
  { \__graph_put_edge:Nnnnn #1 {#2} {#3} {} { } }
\cs_new_protected:Nn \graph_gput_edge:Nnn
  { \__graph_put_edge:Nnnnn #1 {#2} {#3} {} {g} }
\cs_new_protected:Nn \graph_put_edge:Nnnn
  { \__graph_put_edge:Nnnnn #1 {#2} {#3} {#4} { } }
\cs_new_protected:Nn \graph_gput_edge:Nnnn
  { \__graph_put_edge:Nnnnn #1 {#2} {#3} {#4} {g} }
\cs_new_protected:Nn \__graph_put_edge:Nnnnn
  {
    \graph_get_vertex:NnNTF #1 {#2} \l__graph_from_value_tl {
      \graph_get_vertex:NnNTF #1 {#3} \l__graph_to_value_tl {
        \graph_get_edge:NnnNF #1 {#2} {#3} \l_tmpa_tl {
          %%% increment outgoing degree of vertex #2
          %
```

```
          \use:c{prop_#5put:cnf} {\__graph_tl:nnn{graph}{#1}{outdegree}} {#2}
            {\int_eval:n {
                \prop_get:cn {\__graph_tl:nnn{graph}{#1}{outdegree}} {#2} + 1
            }}

          %%% increment incoming degree of vertex #3
          %
          \use:c{prop_#5put:cnf} {\__graph_tl:nnn{graph}{#1}{indegree}} {#3}
            {\int_eval:n {
                \prop_get:cn {\__graph_tl:nnn{graph}{#1}{indegree}} {#3} + 1
            }}
        }

        %%% actually add the edge
        %
        \withargs:VVn \l__graph_from_value_tl \l__graph_to_value_tl {
          \use:c{prop_#5put:cox}
            { \__graph_tl:nnn{graph}{#1}{edge-froms}   }
            { \__graph_key:nn{#2}{#3}                  }
            { \tl_to_str:n{#2}                         }
          \use:c{prop_#5put:cox}
            { \__graph_tl:nnn{graph}{#1}{edge-tos}     }
            { \__graph_key:nn{#2}{#3}                  }
            { \tl_to_str:n{#3}                         }
          \__graph_ptr_new:Nn \l__graph_edge_data_tl {#4}
          \use:c{prop_#5put:coV}
            { \__graph_tl:nnn{graph}{#1}{edge-values}  }
            { \__graph_key:nn{#2}{#3}                  }
            \l__graph_edge_data_tl
          \use:c{prop_#5put:cox}
            { \__graph_tl:nnn{graph}{#1}{edge-triples} }
            { \__graph_key:nn{#2}{#3}                  }
            { {\tl_to_str:n{#2}}
              {\tl_to_str:n{#3}}
              {\l__graph_edge_data_tl}                 }
        }
      }{
        % TODO: Error ('to' vertex doesn't exist)
      }
    }{
      % TODO: Error ('from' vertex doesn't exist)
    }
  }
\cs_generate_variant:Nn \prop_gput:Nnn {cox, coV, cnf}
\cs_generate_variant:Nn \prop_put:Nnn  {cox, coV, cnf}
\cs_generate_variant:Nn \withargs:nnn  {VVn}
\tl_new:N \l__graph_edge_data_tl
\tl_new:N \l__graph_from_value_tl
\tl_new:N \l__graph_to_value_tl
```

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

```
\cs_new_protected:Nn \graph_remove_vertex:Nn
  { \__graph_remove_vertex:Nnn #1 {#2} { } }
```

```
\cs_new_protected:Nn \graph_gremove_vertex:Nn
  { \__graph_remove_vertex:Nnn #1 {#2} {g} }
\cs_new_protected:Nn \__graph_remove_vertex:Nnn
  {
    \graph_get_vertex:NnNT #1 {#2} \l__graph_vertex_data_tl {
      %%% remove outgoing edges
      %
      \graph_map_outgoing_edges_inline:Nnn #1 {#2}
        { \use:c{graph_#3remove_edge:Nnn} #1 {##1} {##2} }

      %%% remove incoming edges
      %
      \graph_map_incoming_edges_inline:Nnn #1 {#2}
        { \use:c{graph_#3remove_edge:Nnn} #1 {##1} {##2} }

      %%% remove the vertex
      %
      \use{prop_#3remove:cn} {\__graph_tl:nnn{graph}{#1}{vertices}}  {#2}
      \use{prop_#3remove:cn} {\__graph_tl:nnn{graph}{#1}{indegree}}  {#2}
      \use{prop_#3remove:cn} {\__graph_tl:nnn{graph}{#1}{outdegree}} {#2}
    }
  }
\cs_generate_variant:Nn \prop_put:Nnn {cnV}
% \tl_new:N \l__graph_vertex_data_tl  % reusing from other function
```

Remove an edge from the graph:

```
\cs_new_protected:Nn \graph_remove_edge:Nnn
  { \__graph_remove_edge:Nnnn #1 {#2} {#3} { } }
\cs_new_protected:Nn \graph_gremove_edge:Nnn
  { \__graph_remove_edge:Nnnn #1 {#2} {#3} {g} }
\cs_new_protected:Nn \__graph_remove_edge:Nnnn {
  \graph_get_edge:NnnNT #1 {#2} {#3} \l__graph_edge_data_tl {
    %%% decrement outdegree of vertex #2
    %
    \use:c{prop_#4put:cnf} {\__graph_tl:nnn{graph}{#1}{outdegree}} {#2}
      {\int_eval:n {
          \prop_get:cn {\__graph_tl:nnn{graph}{#1}{outdegree}} {#2} - 1
      }}

    %%% decrement indegree of vertex #3
    %
    \use:c{prop_#4put:cnf} {\__graph_tl:nnn{graph}{#1}{indegree}} {#3}
      {\int_eval:n {
          \prop_get:cn {\__graph_tl:nnn{graph}{#1}{indegree}} {#3} - 1
      }}

    %%% actually remove edge
    %
    \use:c{prop_#4remove:co}
      { \__graph_tl:nnn{graph}{#1}{edge-froms}   }
      { \__graph_key:nn{#2}{#3}                  }
    \use:c{prop_#4remove:co}
      { \__graph_tl:nnn{graph}{#1}{edge-tos}     }
```

```
      { \__graph_key:nn{#2}{#3}                  }
    \use:c{prop_#4remove:co}
      { \__graph_tl:nnn{graph}{#1}{edge-values}  }
      { \__graph_key:nn{#2}{#3}                  }
    \use:c{prop_#4remove:co}
      { \__graph_tl:nnn{graph}{#1}{edge-triples} }
      { \__graph_key:nn{#2}{#3}                  }
  }
}
\cs_generate_variant:Nn \prop_remove:Nn  {co}
\cs_generate_variant:Nn \prop_gremove:Nn {co}
\cs_generate_variant:Nn \prop_put:Nnn    {cnf}
\cs_generate_variant:Nn \prop_gput:Nnn   {cnf}
%\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`:

```
\cs_new_protected:Nn \graph_put_edges_from:NN
  { \__graph_gput_edges_from:NNn #1 #2 { } }
\cs_new_protected:Nn \graph_gput_edges_from:NN
  { \__graph_gput_edges_from:NNn #1 #2 {g} }
\cs_new_protected:Nn \__graph_gput_edges_from:NNn
  {
    \graph_map_edges_inline:Nn #2 {
      \graph_if_vertex_exist:NnT #1 {##1} {
        \graph_if_vertex_exist:NnT #1 {##2} {
          \graph_gput_edge:Nnnn #1 {##1} {##2} {##3}
        }
      }
    }
  }
```

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.

```
\prg_new_protected_conditional:Nnn \graph_get_vertex:NnN
  {T, F, TF}
  {
    \prop_get:cnNTF { \__graph_tl:nnn {graph} {#1} {vertices} } {#2} #3
      { \tl_set:Nv #3 {#3} \prg_return_true:  }
      {                    \prg_return_false: }
  }
```

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.

```
\prg_new_protected_conditional:Nnn \graph_get_edge:NnnN
  {T, F, TF}
  {
    \prop_get:coNTF
      { \__graph_tl:nnn{graph}{#1}{edge-values} }
      { \__graph_key:nn{#2}{#3}                 }
```

```
      #4
      { \tl_set:Nv #4 {#4} \prg_return_true:  }
      {                     \prg_return_false: }
  }
```

3.9 Graph Conditionals

An expandable test for the existence of a vertex:

```
\prg_new_conditional:Nnn \graph_if_vertex_exist:Nn
  {p, T, F, TF}
  {
    \prop_if_in:cnTF
      { \__graph_tl:nnn {graph} {#1} {vertices} }
      { #2 }
      { \prg_return_true:  }
      { \prg_return_false: }
  }
```

An expandable test for the existence of an edge:

```
\prg_new_conditional:Nnn \graph_if_edge_exist:Nnn
  {p, T, F, TF}
  {
    \prop_if_in:coTF
      { \__graph_tl:nnn {graph} {#1} {edge-values} }
      { \__graph_key:nn{#2}{#3} }
      { \prg_return_true:  }
      { \prg_return_false: }
  }
```

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

```
\cs_new:Npn \graph_if_vertex_can_reach_cycle_p:Nn #1#2
  { \__graph_if_vertex_can_reach_cycle_p:Nnn #1 {#2} {\__graph_empty_set} }
\cs_new:Npn \graph_if_vertex_can_reach_cycle:NnTF #1#2
  { \__graph_if_vertex_can_reach_cycle:NnnTF #1 {#2} {\__graph_empty_set} }
\cs_new:Npn \graph_if_vertex_can_reach_cycle:NnT #1#2
  { \__graph_if_vertex_can_reach_cycle:NnnT #1 {#2} {\__graph_empty_set} }
\cs_new:Npn \graph_if_vertex_can_reach_cycle:NnF #1#2
  { \__graph_if_vertex_can_reach_cycle:NnnF #1 {#2} {\__graph_empty_set} }

\prg_new_conditional:Nnn \__graph_if_vertex_can_reach_cycle:Nnn
  {p, T, F, TF}
  % #1: graph id
  % #2: vertex id
  % #3: visited vertices in 'prop literal' format (internal l3prop)
  {
    \graph_map_outgoing_edges_tokens:Nnn #1 {#2}
      { \__graph_if_vertex_can_reach_cycle:Nnnnn #1 {#3} }
    \prg_return_false:
  }

\cs_new:Nn \__graph_if_vertex_can_reach_cycle:Nnnnn
  % #1: graph id
```

```
  % #2: visited vertices in 'prop literal' format (internal l3prop)
  % #3: start vertex (not used)
  % #4: current vertex
  % #5: edge value (behind ptr, not used)
  {
    \bool_if:nT
      {
        \__graph_set_if_in_p:nn {#2} {#4} ||
        \__graph_if_vertex_can_reach_cycle_p:Nno #1 {#4}
            { \__graph_set_cons:nn {#2} {#4} }
      }
      { \prop_map_break:n {\use_i:nn \prg_return_true:} }
  }
\cs_generate_variant:Nn \__graph_if_vertex_can_reach_cycle_p:Nnn {Nno}
```

Test whether graph `#1` contains any cycles:

```
\prg_new_conditional:Nnn \graph_if_cyclic:N
  {p, T, F, TF}
  % #1: graph id
  {
    \graph_map_vertices_tokens:Nn #1
      { \__graph_if_cyclic:Nnn #1 }
    \prg_return_false:
  }

\cs_new:Nn \__graph_if_cyclic:Nnn
  % #1: graph id
  % #2: vertex id
  % #3: vertex value (not used)
  {
    \bool_if:nT
      { \graph_if_vertex_can_reach_cycle_p:Nn #1 {#2} }
      { \prop_map_break:n {\use_i:nn \prg_return_true:} }
  }
```

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

```
\prg_new_conditional:Nnn \graph_acyclic_if_path_exist:Nnn
  {p, T, F, TF}
  % #1: graph id
  % #2: start vertex
  % #3: end vertex
  {
    \graph_map_outgoing_edges_tokens:Nnn #1 {#2}
      { \__graph_acyclic_if_path_exist:Nnnnn #1 {#3} }
    \prg_return_false:
  }

\cs_new:Nn \__graph_acyclic_if_path_exist:Nnnnn
  % #1: graph id
  % #2: end vertex
  % #3: start vertex (not used)
  % #4: possible end vertex
```

```
  % #5: edge value (behind ptr, do not use)
  {
    \bool_if:nT
      {
        \str_if_eq_p:nn {#4} {#2} ||
        \graph_acyclic_if_path_exist_p:Nnn #1 {#4} {#2}
      }
      { \prop_map_break:n {\use_i:nn \prg_return_true:} }
  }
```

3.10 Querying Information

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

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

Get the number of edges leading into vertex #2:

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

Get the number of edges connected to vertex #2:

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

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.

```
\cs_new:Nn \graph_map_vertices_tokens:Nn {
  \prop_map_tokens:cn
    { \__graph_tl:nnn{graph}{#1}{vertices} }
    { \__graph_map_vertices_tokens_aux:nnv {#2} }
}
\cs_new:Nn \__graph_map_vertices_tokens_aux:nnn
  { #1 {#2} {#3} }
\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.

```
\cs_new:Nn \graph_map_vertices_function:NN {
  \prop_map_tokens:cn
    { \__graph_tl:nnn{graph}{#1}{vertices} }
    { \exp_args:Nnv #2 }
}
```

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.

```
\cs_new_protected:Nn \graph_map_vertices_inline:Nn {
  \withargs (c) [\uniquecsname] [#2] {
    \cs_set:Npn ##1 ####1####2 {##2}
    \graph_map_vertices_function:NN #1 ##1
  }
}
```

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.

```
\cs_new:Nn \graph_map_edges_tokens:Nn {
  \prop_map_tokens:cn
    { \__graph_tl:nnn{graph}{#1}{edge-triples} }
    { \__graph_map_edges_tokens_aux:nnn {#2} }
}
\cs_new:Nn \__graph_map_edges_tokens_aux:nnn
  { \__graph_map_edges_tokens_aux:nnnv {#1} #3 }
\cs_new:Nn \__graph_map_edges_tokens_aux:nnnn
  { #1 {#2} {#3} {#4} }
\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.

```
\cs_new:Nn \graph_map_edges_function:NN {
  \prop_map_tokens:cn
    { \__graph_tl:nnn{graph}{#1}{edge-triples} }
    { \__graph_map_edges_function_aux:Nnn #2 }
}
\cs_new:Nn \__graph_map_edges_function_aux:Nnn
  { \__graph_map_edges_function_aux:Nnnv #1 #3 }
\cs_new:Nn \__graph_map_edges_function_aux:Nnnn
  { #1 {#2} {#3} {#4} }
\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.

```
\cs_new_protected:Nn \graph_map_edges_inline:Nn {
  \withargs (c) [\uniquecsname] [#2] {
    \cs_set:Npn ##1 ####1####2####3 {##2}
    \graph_map_edges_function:NN #1 ##1
  }
}
```

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.

```
\cs_new:Nn \graph_map_incoming_edges_tokens:Nnn {
  % #1: graph
  % #2: base vertex
  % #3: tokens to execute
  \prop_map_tokens:cn
    { \__graph_tl:nnn{graph}{#1}{edge-triples} }
    { \__graph_map_incoming_edges_tokens_aux:nnnn {#2} {#3} }
}
\cs_new:Nn \__graph_map_incoming_edges_tokens_aux:nnnn
  % #1: base vertex
  % #2: tokens to execute
  % #3: edge key
  % #4: edge-triple {from}{to}{value}
  { \__graph_map_incoming_edges_tokens_aux:nnnnv {#1} {#2} #4 }
\cs_new:Nn \__graph_map_incoming_edges_tokens_aux:nnnnn
  % #1: base vertex
  % #2: tokens to execute
  % #3: edge 'from' vertex
  % #4: edge 'to' vertex
  % #5: edge value
  { \str_if_eq:nnT {#1} {#4} { #2 {#3} {#4} {#5} } }
\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.

```
\cs_new:Nn \graph_map_incoming_edges_function:NnN {
  % #1: graph
  % #2: base vertex
  % #3: function to execute
  \prop_map_tokens:cn
    { \__graph_tl:nnn{graph}{#1}{edge-triples} }
    { \__graph_map_incoming_edges_function_aux:nNnn {#2} #3 }
}
\cs_new:Nn \__graph_map_incoming_edges_function_aux:nNnn
  % #1: base vertex
  % #2: function to execute
  % #3: edge key
  % #4: edge-triple {from}{to}{value}
  { \__graph_map_incoming_edges_function_aux:nNnnv {#1} #2 #4 }
\cs_new:Nn \__graph_map_incoming_edges_function_aux:nNnnn
  % #1: base vertex
  % #2: function to execute
  % #3: edge 'from' vertex
  % #4: edge 'to' vertex
  % #5: edge value
  { \str_if_eq:nnT {#1} {#4} { #2 {#3} {#4} {#5} } }
\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.

```
\cs_new_protected:Nn \graph_map_incoming_edges_inline:Nnn {
  % #1: graph
  % #2: base vertex
  % #3: body to execute
  \withargs (c) [\uniquecsname] [#2] [#3] {
    \cs_set:Npn ##1 ####1####2####3 {##3}
    \graph_map_incoming_edges_function:NnN #1 {##2} ##1
  }
}
```

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.

```
\cs_new:Nn \graph_map_outgoing_edges_tokens:Nnn {
  % #1: graph
  % #2: base vertex
  % #3: tokens to execute
  \prop_map_tokens:cn
    { \__graph_tl:nnn{graph}{#1}{edge-triples} }
    { \__graph_map_outgoing_edges_tokens_aux:nnnn {#2} {#3} }
}
\cs_new:Nn \__graph_map_outgoing_edges_tokens_aux:nnnn
  % #1: base vertex
  % #2: tokens to execute
  % #3: edge key (not used)
  % #4: edge-triple {from}{to}{value}
  { \__graph_map_outgoing_edges_tokens_aux:nnnnv {#1} {#2} #4 }
\cs_new:Nn \__graph_map_outgoing_edges_tokens_aux:nnnnn
  % #1: base vertex
  % #2: tokens to execute
  % #3: edge 'from' vertex
  % #4: edge 'to' vertex
  % #5: edge value
  { \str_if_eq:nnT {#1} {#3} { #2 {#3} {#4} {#5} } }
\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.

```
\cs_new:Nn \graph_map_outgoing_edges_function:NnN {
  % #1: graph
  % #2: base vertex
  % #3: function to execute
  \prop_map_tokens:cn
    { \__graph_tl:nnn{graph}{#1}{edge-triples} }
    { \__graph_map_outgoing_edges_function_aux:nNnn {#2} #3 }
}
\cs_new:Nn \__graph_map_outgoing_edges_function_aux:nNnn
  % #1: base vertex
  % #2: function to execute
  % #3: edge key
  % #4: edge-triple {from}{to}{value}
  { \__graph_map_outgoing_edges_function_aux:nNnnv {#1} #2 #4 }
```

```
\cs_new:Nn \__graph_map_outgoing_edges_function_aux:nNnnn
  % #1: base vertex
  % #2: function to execute
  % #3: edge 'from' vertex
  % #4: edge 'to' vertex
  % #5: edge value
  { \str_if_eq:nnT {#1} {#3} { #2 {#3} {#4} {#5} } }
\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.

```
\cs_new_protected:Nn \graph_map_outgoing_edges_inline:Nnn {
  % #1: graph
  % #2: base vertex
  % #3: body to execute
  \withargs (c) [\uniquecsname] [#2] [#3] {
    \cs_set:Npn ##1 ####1####2####3 {##3}
    \graph_map_outgoing_edges_function:NnN #1 {##2} ##1
  }
}
```

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.

```
\cs_new:Nn \graph_map_successors_tokens:Nnn {
  % #1: graph
  % #2: base vertex
  % #3: tokens to execute
  \prop_map_tokens:cn
    { \__graph_tl:nnn{graph}{#1}{edge-triples} }
    { \__graph_map_successors_tokens_aux:Nnnnn #1 {#2} {#3} }
}
\cs_new:Nn \__graph_map_successors_tokens_aux:Nnnnn {
  % #1: the graph
  % #2: base vertex
  % #3: tokens to execute
  % #4: edge key (not used)
  % #5: edge-triple {from}{to}{value}
  \__graph_map_successors_tokens_aux:Nnnnnn #1 {#2} {#3} #5
}
\cs_new:Nn \__graph_map_successors_tokens_aux:Nnnnnn {
  % #1: the graph
  % #2: base vertex
  % #3: tokens to execute
  % #4: edge 'from' vertex
  % #5: edge 'to' vertex
  % #6: ptr to edge value (not used)
  \str_if_eq:nnT {#2} {#4} {
    \__graph_map_successors_tokens_aux:nnv
        {#3} {#5} {\prop_get:cn{\__graph_tl:nnn{graph}{#1}{vertices}}{#5}}
  }
}
```

```
\cs_new:Nn \__graph_map_successors_tokens_aux:nnn {
  % #1: tokens to execute
  % #2: successor key
  % #3: successor value
  #1 {#2} {#3}
}
\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.

```
\cs_new:Nn \graph_map_successors_function:NnN {
  % #1: graph
  % #2: base vertex
  % #3: function to execute
  \prop_map_tokens:cn
    { \__graph_tl:nnn{graph}{#1}{edge-triples} }
    { \__graph_map_successors_function_aux:NnNnn #1 {#2} #3 }
}
\cs_new:Nn \__graph_map_successors_function_aux:NnNnn {
  % #1: the graph
  % #2: base vertex
  % #3: function to execute
  % #4: edge key (not used)
  % #5: edge-triple {from}{to}{value}
  \__graph_map_successors_function_aux:NnNnnn #1 {#2} #3 #5
}
\cs_new:Nn \__graph_map_successors_function_aux:NnNnnn {
  % #1: the graph
  % #2: base vertex
  % #3: function to execute
  % #4: edge 'from' vertex
  % #5: edge 'to' vertex
  % #6: ptr to edge value (not used)
  \str_if_eq:nnT {#2} {#4} {
    \__graph_map_successors_function_aux:Nnv
        #3 {#5} {\prop_get:cn{\__graph_tl:nnn{graph}{#1}{vertices}}{#5}}
  }
}
\cs_new:Nn \__graph_map_successors_function_aux:Nnn {
  % #1: function to execute
  % #2: successor key
  % #3: successor value
  #1 {#2} {#3}
}
\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.

```
\cs_new_protected:Nn \graph_map_successors_inline:Nnn {
  % #1: graph
  % #2: base vertex
```

```
  % #3: body to execute
  \withargs (c) [\uniquecsname] [#2] [#3] {
    \cs_set:Npn ##1 ####1####2####3 {##3}
    \graph_map_successors_function:NnN #1 {##2} ##1
  }
}
```

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).

```
\cs_new_protected:Nn \graph_map_topological_order_tokens:Nn {
  %%% Fill \l__graph_source_vertices with source-nodes and count indegrees
  %
  \prop_clear:N \l__graph_source_vertices
  \graph_map_vertices_inline:Nn #1 {
    \prop_put:Nnf \l__graph_tmp_indegrees_int {##1}
      {\graph_get_indegree:Nn #1 {##1}}
    \int_compare:nT {\graph_get_indegree:Nn #1 {##1} = 0}
      { \prop_put:Nnn \l__graph_source_vertices {##1} {} }
  }

  %%% Main loop
  %
  \bool_until_do:nn {\prop_if_empty_p:N \l__graph_source_vertices} {
    %%% Choose any vertex (\l__graph_topo_key_tl, \l__graph_topo_value_tl)
    %
    \__graph_prop_any_key_pop:NN \l__graph_source_vertices \l__graph_topo_key_tl
    \graph_get_vertex:NVNT #1 \l__graph_topo_key_tl \l__graph_topo_val_tl {
      %%% Run the mapping funtion on the key and value from that vertex
      %
      \withargs:VVn \l__graph_topo_key_tl \l__graph_topo_val_tl
        { #2 {##1} {##2} }

      %%% Deduct one from the counter of all affected nodes
      %%% and add all now-empty vertices to \l__graph_source_vertices
      %
      \graph_map_outgoing_edges_inline:NVn #1 \l__graph_topo_key_tl {
        \prop_put:Nnf \l__graph_tmp_indegrees_int {##2}
          {\int_eval:n {\prop_get:Nn \l__graph_tmp_indegrees_int {##2} - 1}}
        \int_compare:nT {\prop_get:Nn \l__graph_tmp_indegrees_int {##2} = 0} {
          \prop_put:Nnn \l__graph_source_vertices {##2} {}
        }
      }
    }
  }
}
\cs_new_protected:Nn \__graph_prop_any_key_pop:NN {
  \prop_map_inline:Nn #1 {
    \tl_set:Nn #2 {##1}
    \prop_remove:Nn #1 {##1}
    \prop_map_break:n {\use_none:n}
  }
  { \tl_set:Nn #2 {\q_no_value} }
```

```
}
\cs_generate_variant:Nn \withargs:nnn                         {VVn}
\cs_generate_variant:Nn \graph_map_outgoing_edges_inline:Nnn {NVn}
\cs_generate_variant:Nn \prop_put:Nnn                         {Nnf}
\cs_generate_variant:Nn \graph_get_vertex:NnNT                {NVNT}
\prop_new:N \l__graph_source_vertices
\prop_new:N \l__graph_tmp_indegrees_int
\tl_new:N \l__graph_topo_key_tl
\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).

```
\cs_new:Nn \graph_map_topological_order_function:NN
  { \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).

```
\cs_new_protected:Nn \graph_map_topological_order_inline:Nn {
  \withargs (c) [\uniquecsname] [#2] {
    \cs_set:Npn ##1 ####1####2 {##2}
    \graph_map_topological_order_function:NN #1 ##1
  }
}
```

3.12 Transforming Graphs

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

```
\cs_new_protected:Nn \graph_set_transitive_closure:NN
  { \__graph_set_transitive_closure:NNNnn #1 #2 \use_none:nnn {} { } }
\cs_new_protected:Nn \graph_gset_transitive_closure:NN
  { \__graph_set_transitive_closure:NNNnn #1 #2 \use_none:nnn {} {g} }
\cs_new_protected:Nn \graph_set_transitive_closure:NNNn
  { \__graph_set_transitive_closure:NNNnn #1 #2 #3 {#4} { } }
\cs_new_protected:Nn \graph_gset_transitive_closure:NNNn
  { \__graph_set_transitive_closure:NNNnn #1 #2 #3 {#4} {g} }
\cs_new_protected:Nn \__graph_set_transitive_closure:NNNnn
  % #1: target
  % #2: source
  % #3: combination function with argspec :nnn
  % #4: default 'old' value
  {
    \use:c{graph_#5set_eq:NN} #1 #2

    \cs_set:Nn \__graph_edge_combinator:nnn
      { \exp_not:n { #3 {##1} {##2} {##3} } }
    \cs_generate_variant:Nn \__graph_edge_combinator:nnn {VVV}

    \graph_map_vertices_inline:Nn #2 {
      \graph_map_vertices_inline:Nn #2 {
```

```
        \graph_get_edge:NnnNT #2 {##1} {####1} \l__graph_edge_value_i_tl {
          \graph_map_vertices_inline:Nn #2 {
            \graph_get_edge:NnnNT #2 {####1} {########1}
                \l__graph_edge_value_ii_tl {
              \graph_get_edge:NnnNF #1 {##1} {########1}
                  \l__graph_edge_value_old_tl
                { \tl_set:Nn \l__graph_edge_value_old_tl {#4} }
              \exp_args:NNx \tl_set:No \l__graph_edge_value_new_tl {
                \__graph_edge_combinator:VVV
                  \l__graph_edge_value_i_tl
                  \l__graph_edge_value_ii_tl
                  \l__graph_edge_value_old_tl
              }
              \use:c{graph_#5put_edge:NnnV} #1 {##1} {########1}
                  \l__graph_edge_value_new_tl
            }
          }
        }
      }
    }
  }
\cs_generate_variant:Nn \graph_put_edge:Nnnn  {NnnV}
\cs_generate_variant:Nn \graph_gput_edge:Nnnn {NnnV}
\cs_generate_variant:Nn \tl_to_str:n          {o}
\tl_new:N \l__graph_edge_value_i_tl
\tl_new:N \l__graph_edge_value_ii_tl
\tl_new:N \l__graph_edge_value_old_tl
\tl_new:N \l__graph_edge_value_new_tl
```

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

```
\cs_new_protected:Nn \graph_set_transitive_reduction:NN
  { \__graph_set_transitive_reduction:NNn #1 #2 { } }
\cs_new_protected:Nn \graph_gset_transitive_reduction:NN
  { \__graph_set_transitive_reduction:NNn #1 #2 {g} }
\cs_new_protected:Nn \__graph_set_transitive_reduction:NNn
  % #1: target
  % #2: source
  {
    \use:c{graph_#3set_eq:NN} #1 #2
    \graph_map_vertices_inline:Nn #2 {
      \graph_map_vertices_inline:Nn #2 {
        \graph_get_edge:NnnNT #2 {##1} {####1} \l_tmpa_tl {
          \graph_map_vertices_inline:Nn #2 {
            \graph_get_edge:NnnNT #2 {####1} {########1} \l_tmpa_tl {
              \use:c{graph_#3remove_edge:Nnn} #1 {##1} {########1}
            }
          }
        }
      }
    }
  }
```

3.13 Displaying Graphs

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

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

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

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

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

```
\keys_define:nn {lt3graph-display} {
  row_keys .bool_set:N  = \l__graph_display_row_keys_bool,
  row_keys .initial:n   = {true},
  row_keys .default:n   = {true},

  vertex_vals .bool_set:N  = \l__graph_display_vertex_vals_bool,
  vertex_vals .initial:n   = {true},
  vertex_vals .default:n   = {true},

  row_keys_format      .tl_set:N  = \l__graph_format_row_keys_tl,
  row_keys_format      .initial:n = \textbf,
  row_keys_format      .value_required:,

  col_keys_format      .tl_set:N  = \l__graph_format_col_keys_tl,
  col_keys_format      .initial:n = \textbf,
  col_keys_format      .value_required:,

  vertex_vals_format   .tl_set:N  = \l__graph_format_vertex_vals_tl,
  vertex_vals_format   .initial:n = \use:n,
  vertex_vals_format   .value_required:,

  edge_vals_format     .tl_set:N  = \l__graph_format_edge_vals_tl,
  edge_vals_format     .initial:n = \use:n,
  edge_vals_format     .value_required:,

  edge_diagonal_format .tl_set:N  = \l__graph_format_edge_diagonal_tl,
  edge_diagonal_format .initial:n = \cellcolor{black!30!white},
  edge_diagonal_format .value_required:,

  edge_direct_format   .tl_set:N  = \l__graph_format_edge_direct_tl,
  edge_direct_format   .initial:n = \cellcolor{green},
  edge_direct_format   .value_required:,

  edge_transitive_format .tl_set:N  = \l__graph_format_edge_transitive_tl,
  edge_transitive_format .initial:n = \cellcolor{green!40!yellow}\tiny(tr),
  edge_transitive_format .value_required:,

  edge_none_format     .tl_set:N  = \l__graph_format_edge_none_tl,
  edge_none_format     .initial:n = {},
  edge_none_format     .value_required:
```

```
}
```

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.

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

We process those options passed with #2:

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

We populate the top row of the table:

```
  \tl_put_right:Nn \l__graph_table_content_tl {\hline}
  \seq_clear:N \l__graph_row_seq
  \bool_if:NT \l__graph_display_row_keys_bool
      { \seq_put_right:Nn \l__graph_row_seq {}
        \tl_put_right:Nn \l__graph_table_colspec_tl {|r|} }
  \bool_if:NT \l__graph_display_vertex_vals_bool
      { \seq_put_right:Nn \l__graph_row_seq {}
        \tl_put_right:Nn \l__graph_table_colspec_tl {|c|} }
  \graph_map_vertices_inline:Nn #1 {
    \tl_put_right:Nn \l__graph_table_colspec_tl {|c}
    \seq_put_right:Nn \l__graph_row_seq
        { { \l__graph_format_col_keys_tl {##1} } }
  }
  \tl_put_right:Nn \l__graph_table_colspec_tl {|}
  \tl_put_right:Nx \l__graph_table_content_tl
      { \seq_use:Nn \l__graph_row_seq {&} }
  \tl_put_right:Nn \l__graph_table_content_tl
      { \\\hline\hline }
```

We populate the remaining rows:

```
  \graph_map_vertices_inline:Nn #1 {
    \seq_clear:N \l__graph_row_seq
    \bool_if:NT \l__graph_display_row_keys_bool {
      \seq_put_right:Nn \l__graph_row_seq
        { { \l__graph_format_row_keys_tl {##1} } } }
    \bool_if:NT \l__graph_display_vertex_vals_bool {
      \seq_put_right:Nn \l__graph_row_seq
        { { \l__graph_format_vertex_vals_tl {##2} } } }
    \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:

```
      \graph_get_edge:NnnNTF #1 {##1} {####1} \l_tmpa_tl {
        \quark_if_no_value:VF \l_tmpa_tl {
          \tl_set_eq:NN \l__graph_cell_content_tl \l_tmpa_tl
          \tl_set:Nf \l__graph_cell_content_tl
              { \exp_args:NV \l__graph_format_edge_direct_tl
                             \l__graph_cell_content_tl } }
```

```
      }{\graph_acyclic_if_path_exist:NnnTF #1 {##1} {####1} {
        \tl_set_eq:NN \l__graph_cell_content_tl
            \l__graph_format_edge_transitive_tl
      }{
        \tl_set_eq:NN \l__graph_cell_content_tl
            \l__graph_format_edge_none_tl
      }}
```

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

```
      \str_if_eq:nnT {##1} {####1} {
        \tl_set:Nf \l__graph_cell_content_tl
            { \exp_args:NV \l__graph_format_edge_diagonal_tl
                           \l__graph_cell_content_tl } }
```

Tertiary formatting is applied to all vertex value cells:

```
      \tl_set:Nf \l__graph_cell_content_tl
          { \exp_args:NV \l__graph_format_edge_vals_tl
                         \l__graph_cell_content_tl }
```

We can now add the cell to the row sequence:

```
      \seq_put_right:NV \l__graph_row_seq \l__graph_cell_content_tl
```

```
    }
```

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

```
    \tl_put_right:Nx \l__graph_table_content_tl { \seq_use:Nn \l__graph_row_seq {&} }
    \tl_put_right:Nn \l__graph_table_content_tl {\\\hline}
```

```
  }
```

Finally, we print the table itself:

```
  \withargs:VVn \l__graph_table_colspec_tl \l__graph_table_content_tl
    { \begin{tabular}{##1}##2\end{tabular} }
```

```
  \group_end:
}
```

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

```
\cs_generate_variant:Nn \quark_if_no_value:nF {VF}
\cs_generate_variant:Nn \withargs:nnn         {VVn}
\tl_new:N \l__graph_table_colspec_tl
\tl_new:N \l__graph_table_content_tl
\tl_new:N \l__graph_cell_content_tl
\bool_new:N \l__graph_table_skipfirst_bool
\seq_new:N \l__graph_row_seq
```

```
871 }
```

Change History

0.0.1
General: initial version 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

U

W