Project: ISO JTC1/SC22/WG21: Programming Language C++
Doc No: WG21 D9903r0 — Graph Library Operators
Date: 2024-02-11
Reply to: Phil Ratzloff (phil.ratzloff@sas.com),
Andrew Lumsdaine (lumsdaine@gmail.com)

Contributors: Richard Dosselmann (University of Regina)
Michael Wong (Codeplay)
Matthew Galati (Amazon)
Jens Maurer
Jesun Firoz
Kevin Deweese
Muhammad Osama (AMD, Inc)

Audience: SG19, SG14, SG6, LEWG, LWG
Source: github.com/stdgraph/graph-v2
Prev. Version: www.wg21.link/P1709r5

Contents

Chapter1 Getting Started

This paper is one of several interrelated proposals related to a Graph Library proposal that have been broken out for easier consumption. The following table describes all the related papers.

Paper	Status	Description
P1709	Inactive	Original proposal, now broken into the following papers.
P9901	Active	**Graph Library Overview and Introduction**, describing the big picture of what we are proposing, and theortical background of graphs in general.
P9902	Active	**Graph Library Algorithms**, covering the initial algorithms as well as the ones we'd like to see in the future.
P9903	Active	**Graph Library Operators** includes useful utility functions when working with graphs.
P9904	Active	**Graph Library Views** including helpful views for traversing a graph.
P9905	Active	**Graph Library Container Inferface** is the core interface used for accessing the underlying graph data structure.
P9906	Active	**Graph Library Container** describing the high-performance `compressed_graph` container, based on a Compressed Sparse Row sparse matrix layout.
P9907	Active	**Graph Library Adaptors** containing useful utilities to convert graphs to different forms.

Table 1.1 — Graph Library Papers

1.1 Revision History

D9903r0

— Split from P1709r5. Added *Getting Started* chapter.

1.2 Introduction

Basic characteristics of the algorithms shown below are summarized in tables of the following form:

Complexity $\mathcal{O}(\|E\| + \|V\|)$	**Throws?** No **Multi-edge?** No	**Cycles?** No **Directed?** Yes

The parts of the table have the following meaning:

— **Complexity** The complexity of the algorithm based on the number of vertices (V) and edges (E).

— **Throws?** Will the algorithm throw at all? If so, look at the *Throws* section after the function prototypes for details.

— **Multi-edge?** Does the algorithm act as expected if more than one edge with the same direction exists between the same two vertices?

— **Cycles?** Does the algorithm act act as expected if a vertex (or edge) is part of a cycle?

— **Directed?** Is the algorithm only for directed graphs, or can it also be used for undirected graphs that have complimentary edges, with different directions, between two vertices.

[PHIL: The Directed? section needs work.]

Chapter2 Operators

2.1 Degree

Degree of a vertex in graph theory refers to the number of edges that are incident to the vertex. Although the proposal provides customization point interface to access a degree of an individual vertex, a degree operator allows one to build a range of degrees of all vertices within a graph.

```
template <index_adjacency_list G,
          ranges::random_access_range D>
requires is_integral_v<ranges::range_value_t<P>>
void degrees(const G& graph, D& degrees_per_vertex) {}
```

1 *Preconditions:*

(1.1) — `graph` is an `index_adjacency_list`, which can be directed or undirected. [MUHAMMAD: Support other graph types as well?]

2 *Effects:* Output iterator `degrees_per_vertex` contains the degree of each vertex in the graph, accessible through `degrees_per_vertex[uid]`, where uid is the `vertex_id`. The caller must assure `size(degrees_per_vertex)>= size(vertices(graph))`. *Complexity:* $\mathcal{O}(|V|)$ where V is the number of vertices in the graph.

2.2 Sort

Sort operator comes in many different variants, allows to sort the input graph based on the source vertex, target vertex, the degree of the vertex or by the weight on the edges. Also relevant, `topological_sort`, which returns a nonunique permutation of the vertices of a directed acyclic graph such that an edge from u to v implies that u appears before v in the topological sort order.

2.2.1 Sort by Source Vertex

```
template <index_adjacency_list G>
void sort_by_source(G&& graph) {}
```

1 *Preconditions:*

(1.1) — `graph` is an `index_adjacency_list`, which can be directed or undirected. The same API extends for `edgelist`.

2 *Effects:* `graph`, which is now sorted by the source vertices in the graph.

2.2.2 Sort by Target Vertex

```
template <index_adjacency_list G>
void sort_by_target(G&& graph) {}
```

1 *Preconditions:*

(1.1) — `graph` is an `index_adjacency_list`, which can be directed or undirected. The same API extends for `edgelist`.

2 *Effects:* `graph`, which is now sorted by the target vertices in the graph.

2.2.3 Sort by Degree

```
template <index_adjacency_list G>
void sort_by_degree(G&& graph) {}
```

1 *Preconditions:*

(1.1) — `graph` is an `index_adjacency_list`, which can be directed or undirected. The same API extends for `edgelist`.

2 *Effects:* `graph`, which is now sorted by the vertex degree.

2.2.4 Sort by Edge Weight

```
template <index_adjacency_list G, class W>
requires weight_function<W, edge_t<G>>
void sort_by_weight(G&& graph, W&& w) {}
```

1 *Preconditions:*

(1.1) — `graph` is an `index_adjacency_list`, which can be directed or undirected. The same API extends for `edgelist`.

(1.2) — `w` The edge value function.

2 *Effects:* `graph`, which is now sorted by the edge weight.

2.3 Relabel

Relabels the vertices of the graph according to a given mapping or to integers.

```
template <index_adjacency_list G>
void relabel(G&& graph, const std::map</*?*/, /*?*/>& mapping) {}
```

1 *Preconditions:*

(1.1) — `graph` is an `index_adjacency_list`, which can be directed or undirected. The same API extends for `edgelist`.

(1.2) — `mapping` is a `std::map`, a dictionary with old labels as keys and new labels as values. If a partial mapping is provided (i.e. `mapping.size()< size(vertices(graph))`), then only the specified vertices are relabeled.

2 *Effects:* `graph`: the vertices of the graph are relabeled starting based on the mapping provided.

```
template <index_adjacency_list G>
void relabel_to_integers(G&& graph) {}
```

3 *Preconditions:*

(3.1) — `graph` is an `index_adjacency_list`, which can be directed or undirected. The same API extends for `edgelist`.

4 *Effects:* `graph`: the vertices of the graph are relabeled starting from 0 to `size(vertices(graph))`.

2.4 Transpose

Transpose returns a graph with edges reversed. For an adjacency graph, it is obtained by switching the outer range of vertices with an inner range of incidence edges on each vertex.

```
template <index_adjacency_list G>
void transpose(G&& graph) {}
```

1 *Preconditions:*

(1.1) — `graph` is an `index_adjacency_list`, which can be directed or undirected. The same API extends for `edgelist`.

2 *Effects:* `graph`: the graph is transposed such that the edges for each vertex are reversed.

2.5 Join

Combining of two graphs based on common vertices. The join operation (or Sparse General Matrix-Matrix Multiplication) is a useful operators for implementing graph algorithms. The result would be a new graph where the edges are determined by the multiplication of the adjacency containers.

```
template <index_adjacency_list A, index_adjacency_list B,
        index_adjacency_list C>
```

1 *Preconditions:*

(1.1) — `a`, `b` and `c` are `index_adjacency_list` graphs, which can be directed or undirected.

2 *Effects:* `c`: the resulting graph where the entry (u, v) in the result represents the number of common neighbors between vertex u in Graph A and vertex v in Graph B.

Acknowledgements

Phil Ratzloff's time was made possible by SAS Institute.

Portions of *Andrew Lumsdaine's* time was supported by NSF Award OAC-1716828 and by the Segmented Global Address Space (SGAS) LDRD under the Data Model Convergence (DMC) initiative at the U.S. Department of Energy's Pacific Northwest National Laboratory (PNNL). PNNL is operated by Battelle Memorial Institute under Contract DE-AC06-76RL01830.

Michael Wong's work is made possible by Codeplay Software Ltd., ISOCPP Foundation, Khronos and the Standards Council of Canada.

The authors additionally thank the members of SG19 and SG14 study groups for their invaluable input.