

Graph Library: Comparison

Document #: **D3337r0**
Date: 2024-09-12
Project: Programming Language C++
Audience: Library Evolution
SG19 Machine Learning
SG14 Game, Embedded, Low Latency
SG6 Numerics
Revises: (none)

Reply-to: Phil Ratzloff (SAS Institute)
phil.ratzloff@sas.com
Andrew Lumsdaine
lumsdaine@gmail.com

Contributors: Kevin Deweese
Muhammad Osama (AMD, Inc)
Jesun Firoz
Michael Wong (Intel)
Jens Maurer
Richard Dosselmann (University of Regina)
Matthew Galati (Amazon)
Guy Davidson (Creative Assembly)
Oliver Rosten

1 Getting Started

This paper is one of several interrelated papers for a proposed Graph Library for the Standard C++ Library. The Table 1 describes all the related papers.

Paper	Status	Description
P1709	Inactive	Original proposal, now separated into the following papers.
P3126	Active	Overview , describes the big picture of what we are proposing.
P3127	Active	Background and Terminology provides the motivation, theoretical background, and terminology used across the other documents.
P3128	Active	Algorithms covers the initial algorithms as well as the ones we'd like to see in the future.
P3129	Active	Views has helpful views for traversing a graph.
P3130	Active	Graph Container Interface is the core interface used for uniformly accessing graph data structures by views and algorithms. It is also designed to easily adapt to existing graph data structures.
P3131	Active	Graph Containers describes a proposed high-performance compressed_graph container. It also discusses how to use containers in the standard library to define a graph, and how to adapt existing graph data structures.
P3337	Active	Comparison to other graph libraries on performance and usage syntax.

Table 1: Graph Library Papers

Reading them in order will give the best overall picture. If you're limited on time, you can use the following guide to focus on the papers that are most relevant to your needs.

Reading Guide

- If you're **new to the Graph Library**, we recommend starting with the *Overview* ([P3126](#)) paper to understand the focus and scope of our proposals. You'll also want to check out it stacks up against other graph libraries in performance and usage syntax in the *Comparison* ([P3337](#)) paper.
- If you want to **understand the terminology and theoretical background** that underpins what we're doing, you should read the *Background and Terminology* ([P3127](#)) paper.
- If you want to **use the algorithms**, you should read the *Algorithms* ([P3128](#)) and *Graph Containers* ([P3131](#)) papers. You may also find the *Views* ([P3129](#)) and *Graph Container Interface* ([P3130](#)) papers helpful.
- If you want to **write new algorithms**, you should read the *Views* ([P3129](#)), *Graph Container Interface* ([P3130](#)), and *Graph Containers* ([P3131](#)) papers. You'll also want to review existing implementations in the reference library for examples of how to write the algorithms.
- If you want to **use your own graph data structures**, you should read the *Graph Container Interface* ([P3130](#)) and *Graph Containers* ([P3131](#)) papers.

2 Revision History

D3337r0

- New paper comparing the Graph Library to the NWGraph and Boost Graph Libraries on performance and usage syntax.

3 Naming Conventions

Table 2 shows the naming conventions used throughout the Graph Library documents.

Template Parameter	Type Alias	Variable Names	Description
G			Graph
	<code>graph_reference_t<G></code>	<code>g</code>	Graph reference
GV		<code>val</code>	Graph Value, value or reference
EL		<code>el</code>	Edge list
V	<code>vertex_t<G></code> <code>vertex_reference_t<G></code>	<code>u,v,x,y</code>	Vertex Vertex reference. <code>u</code> is the source (or only) vertex. <code>v</code> is the target vertex.
VId	<code>vertex_id_t<G></code>	<code>uid,vid,seed</code>	Vertex id. <code>uid</code> is the source (or only) vertex id. <code>vid</code> is the target vertex id.
VV	<code>vertex_value_t<G></code>	<code>val</code>	Vertex Value, value or reference. This can be either the user-defined value on a vertex, or a value returned by a function object (e.g. <code>VVF</code>) that is related to the vertex.
VR	<code>vertex_range_t<G></code>	<code>ur,vr</code>	Vertex Range
VI	<code>vertex_iterator_t<G></code>	<code>ui,vi</code>	Vertex Iterator. <code>ui</code> is the source (or only) vertex.
		<code>first,last</code>	<code>vi</code> is the target vertex.
VVF		<code>vvf</code>	Vertex Value Function: <code>vvf(u) → vertex value</code> , or <code>vvf(uid) → vertex value</code> , depending on requirements of the consume algorithm or view.
VProj		<code>vproj</code>	Vertex info projection function: <code>vproj(x) → vertex_info<VId,VV></code> .
	<code>partition_id_t<G></code>	<code>pid</code>	Partition id.
		<code>P</code>	Number of partitions.
PVR	<code>partition_vertex_range_t<G></code>	<code>pur,pvr</code>	Partition vertex range.
E	<code>edge_t<G></code> <code>edge_reference_t<G></code>	<code>uv,vw</code>	Edge Edge reference. <code>uv</code> is an edge from vertices <code>u</code> to <code>v</code> . <code>vw</code> is an edge from vertices <code>v</code> to <code>w</code> .
EV	<code>edge_value_t<G></code>	<code>val</code>	Edge Value, value or reference. This can be either the user-defined value on an edge, or a value returned by a function object (e.g. <code>EVF</code>) that is related to the edge.
ER	<code>vertex_edge_range_t<G></code>		Edge Range for edges of a vertex
EI	<code>vertex_edge_iterator_t<G></code>	<code>uvi,vwi</code>	Edge Iterator for an edge of a vertex. <code>uvi</code> is an iterator for an edge from vertices <code>u</code> to <code>v</code> . <code>vwi</code> is an iterator for an edge from vertices <code>v</code> to <code>w</code> .
EVF		<code>evf</code>	Edge Value Function: <code>evf(uv) → edge value</code> , or <code>evf(eid) → edge value</code> , depending on the requirements of the consuming algorithm or view.
EProj		<code>eproj</code>	Edge info projection function: <code>eproj(x) → edge_info<VId,Sourced,EV></code> .

Table 2: Naming Conventions for Types and Variables

4 Syntax Comparison

<pre> using namespace boost; using G = compressed_sparse_row_graph<directedS, no_property, no_property>; using VId = graph_traits<G>::vertex_descriptor; G g; //populate g vector<VId> parents(num_vertices(g)); auto vis = make_bfs_visitor(make_pair(record_predecessors(parents.begin(), on_tree_edge()))); breadth_first_search(g, vertex(0, g), visitor(vis)); </pre>	<pre> using namespace std; using namespace graph; using G = container::compressed_graph<void, void, void, uint32_t, uint32_t>; using VId = vertex_id_t<G>; G g; // populate g vector<VId> parents(size(vertices(g))); auto bfs = edges_breadth_first_search_view<G,void ,true>(g, 0); for (auto&& [uid, vid, uv] : bfs) { parents[vid] = uid; } </pre>
---	---

Figure 1: BreadthFirst Search Syntax Comparison

<pre> using namespace std; using namespace boost; using G = compressed_sparse_row_graph<directedS, no_property, no_property>; G g; //populate g vector<size_t> component(N); size_t num = connected_components(g, &component [0]); </pre>	<pre> using namespace std; using namespace graph; using G = container::compressed_graph<void, void, void, uint32_t, uint32_t>; G g; //populate g vector<size_t> component(size(vertices(g))); connected_components(g, component); size_t num = ranges::max_element(component) + 1; </pre>
---	---

Figure 2: Connected Components Syntax Comparison

5 Performance Comparison

To evaluate the performance of this proposed library, we compare its reference implementation (stdgraph) against BGL and NWGraph on a subset of the GAP Benchmark Suite[?]. This comparison includes four of the five GAP algorithms that are in the tier 1 algorithm list of this proposal: triangle counting (TC), weak connected components (CC), breadth-first search (BFS), and single-source shortest paths (SSSP). Table 3 summarizes the graphs specified by the GAP benchmark. These graphs were chosen to be large but still fit on shared memory machines and have edge counts in the billions. We compare to BGL because it the commonly used sequential C++ graph library as described above. NWGraph was implemented with many of the ideas of this proposal in mind, and we expect very similar performance between NWGraph and this reference implementation.

The NWGraph authors published a similar comparison to BGL[?] in which they demonstrated performance improvement of NWGraph over BGL. To simplify experimental setup, we rerun these new experiments using the

```

using namespace std;
using namespace boost;

using G =
    compressed_sparse_row_graph<directedS,
        no_property, property<edge_weight_t, int>>;
using VId = graph_traits<G>::vertex_descriptor;

G g;
//populate g

vector<VId> p(num_vertices(g)); //predecessors
vector<int> d(num_vertices(g)); //distances
property_map< graph_t, edge_weight_t >::type
    weightmap = get(edge_weight, g);

dijkstra_shortest_paths(g, vertex(0, g),
    predecessor_map(make_iterator_property_map( p
        .begin(), get(vertex_index, g))).distance_map(
        make_iterator_property_map(d.begin(), get(
            vertex_index, g))));

```

```

using namespace std;
using namespace graph;

using G = container::compressed_graph<int, void,
    void, uint32_t, uint32_t>;

G g;
//populate g

vector<int> d(size(vertices(g)), 0); //distances
auto weight_fn = [&g](graph::edge_reference_t<
    graph_type> uv) -> ValType {
    return edge_value(g, uv);
};

dijkstra_shortest_distances(g, 0, d, weight_fn);

```

Figure 3: Single Source Shortest Paths (Dijkstra) Syntax Comparison

Name	Description	#Vertices (M)	#Edges (M)	Degree Distribution	(Un)directed	References
road	USA road network	23.9	57.7	bounded	undirected	
Twitter	Twitter follower links	61.6	1,468.4	power	directed	
web	Web crawl of .sk domain	50.6	1,930.3	power	directed	
kron	Synthetic graph	134.2	2,111.6	power	undirected	
urand	Uniform random graph	134.2	2,147.5	normal	undirected	

Table 3: Summary of GAP Benchmark Graphs

same machine used in[?], (compute nodes consisting of two Intel® Xeon® Gold 6230 processors, each with 20 physical cores running at 2.1 GHz, and 188GB of memory per processor). NWGraph and stdgraph were compiled with gcc 13.2 using -Ofast -march=native compilation flags.

Even though NWGraph contains an implementation of Dijkstra, the SSSP results in [?] were based on delta-stepping. For this comparison, stdgraph and NWgraph both use Dijkstra. The NWGraph and stdgraph implementation of CC is based on the Afforest [?] algorithm. While BFS and SSSP implementations are very similar for NWGraph and stdgraph, the latter contains support for event-based visitors, and it is important to make sure this does not incur a performance penalty. Table 4 summarizes our GAP benchmark results for stdgraph compared to BGL and NWGraph.

<pre>using namespace boost; using G = compressed_sparse_row_graph<directedS, no_property, no_property>; using VId = graph_traits<G>::vertex_descriptor; G g; //populate g size_t count = 0; for(size_t i = 0; i < N; i++) { VId cur = vertex(i, g); count += num_triangles_on_vertex(g, cur); } count /= 6;</pre>	<pre>using namespace graph; using G = container::compressed_graph<void, void, void, uint32_t, uint32_t>; G g; //populate g size_t count; count = triangle_count(g);</pre>
---	---

Figure 4: TC Syntax Comparison

Algorithm	Library	road	twitter	kron	web	urand
BFS	BGL	1.09s	12.11s	54.80s	5.52s	73.26s
	NWGraph	0.91s	11.25s	38.86s	2.37s	64.63s
	stdgraph	1.39s	8.54s	16.34s	3.52s	62.75s
CC	BGL	1.36s	21.96s	81.18s	6.64s	134.23s
	NWGraph	1.05s	3.77s	10.16s	3.04s	36.59s
	stdgraph	0.78s	2.81s	8.37s	2.23s	33.75s
SSSP	BGL	4.03s	47.89s	167.20s	28.29s	OOM
	NWGraph	3.63s	109.37s	344.12s	35.58s	400.23s
	stdgraph	4.22s	79.75s	211.37s	33.87s	493.15s
TC	BGL	1.34s	>24H	>24H	>24H	4425.54s
	NWGraph	0.41s	1327.63s	6840.38s	131.47s	387.53s
	stdgraph	0.17s	459.08s	2357.95s	50.04s	191.36s

Table 4: GAP Benchmark Performance: Time for GAP benchmark algorithms is shown for Boost Graph Library, NWGraph, and this proposal's reference implementation (stdgraph)

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.

Muhammad Osama's time was made possible by Advanced Micro Devices, Inc.

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