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

Table 2: Naming Conventions for Types and Variables

For the algorithms in this paper, the reference implementation of the proposed graph library is referred to as **graph-v2** [1]. A recent library that this implementation is based on is referred to as **NWGraph** [2, 3]. **BGL** is used to refer to algorithms using the Boost Graph Library [4].

4 Syntax Comparison

In this section, we provide a usage syntax comparison of several graph algorithms in Tier 1 of P3128 against the equivalent implementations in **BGL** and the more recent **NWGraph**. These algorithms are breadth-first search (BFS, Figure 1), connected components (CC, Figure 2), single source shortest paths (SSSP, Figure 3), and triangle counting (TC, Figure 4). We take these algorithms from the GAP Benchmark Suite [5]. We defer to later sections any discussion of underlying implementation details and resulting performance.

Unlike **BGL**, **graph-v2** does not specify edge direction as a graph property. If a graph in **graph-v2** implemented by `container::compressed_graph` is undirected, then it will contain distinct edges in both directions. **BGL** has a `boost::graph::undirectedS` property which can be used in the `boost::graph::adjacency_matrix` class to specify an undirected graph, but not in the `boost::graph::compressed_sparse_row_graph` class. Thus in Figures 1-4, the **BGL** graph type always includes `boost::graph::directedS` . Similar to **graph-v2**, undirected graphs must contain the edges in both directions.

Intermediate data structures (e.g., edge lists) will be needed to construct the compressed graph structures. In order to focus on the differences in algorithm syntax, we omit code which populates the graph data structures. In the following subsections, we address the syntax differences for each of these algorithms.

```cpp
using namespace std;
using namespace boost;

using G = compressed_sparse_row_graph<
            directedS, no_property, no_property>;
using Vertex = graph_traits<G>::vertex_descriptor;

G g;
//populate g

vector<Vertex> 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));
```

(a) **BGL**

```cpp
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;
}
```

(b) **graph-v2**

Figure 1: Breadth-First Search Syntax Comparison

4.1 Breadth-First Search

Figure 1 compares the simplest **BGL** BFS visitor against the range-based-for loop implementation of **graph-v2**. BFS is often described as a graph algorithm, though a BFS traversal by itself does not actually perform any task. In reality, it is a data access pattern which specifies an order vertices and edges should be processed by some higher level algorithm. **BGL** provides a very customizable interface to this data access pattern through the use of visitors which allows users to customize function calls during BFS events. For example `discover_vertex` is

```cpp
using namespace std;
using namespace boost;

using G =
  compressed_sparse_row_graph<
     directedS, no_property, no_property>;

G g;
//populate g

vector<size_t> c(num_vertices(g)); //components
int num_cmps = connected_components(g, &c[0]);
```

(a) **BGL**

```cpp
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> c(size(vertices(g))); //components
int num_cmps = connected_components(g, c);
```

(b) **graph-v2**

Figure 2: Connected Components Syntax Comparison

called when a vertex is encountered for the first time; `examine_vertex` is called when a vertex is popped from the queue; `examine_edge` is called on each edge of a vertex when it is discovered, etc. Figure 1(a) demonstrates the usage of a BFS visitor `record_predecessors` which is called upon event `on_tree_edge` during BFS traversal to store the parent node of every discovered vertex.

This capability is very powerful but often cumbersome if the BFS traversal simply requires vertex and edge access upon visiting. For this reason **graph-v2** provides a simple, range-based-for loop BFS traversal called a view. Figure 1(b) demonstrates how the visited edge `uv` and incident vertices `uid` and `vid` are exposed to the library user to store the parent information explicitly. The authors of this proposal acknowledge that some power users still want the full customization provided by visitors, and we plan to add them to this proposal.

Also note **BGL** often requires the use of vertex descriptors to uniquely identify vertices, as shown by the `graph_traits<G>::vertex_descriptor` type in Figure 1(a). Algorithms written using **graph-v2** use a unique vertex id, as shown by the `vertex_id_t<G>` type in Figure 1(b). This same difference is seen in the algorithms that follow.

4.2 Connected Components

There is very little difference in the connected component interfaces.

[Scott: There is at least one difference. The requirements on the container that holds the component information. **BGL** seems to require a C-array or at the very least a pointer like thing to contiguous memory. What exactly does **graph-v2** require? What is the concept? Is it more flexible than the BGL interface?]

4.3 Single Source Shortest Paths

Of the four algorithms discussed here, only SSSP makes use of an edge property associated with the input graph, the distance used to compute shortest paths. The algorithm computes for every vertex (1) a distance from the start vertex, and (2) a predecessor vertex along the shortest path. In Figure 3 we see that **BGL** requires property maps to lookup edge and vertex properties. These property maps are tightly coupled with the graph data structures. With **graph-v2**, we propose properties be stored external to the graph. For edge properties we provide a weight lambda function to the algorithm to lookup distance from the `edge_reference_t` .

4.4 Triangle Counting

BGL does not provide a triangle counting algorithm similar to the one proposed in **graph-v2**. The code example in Figure 4(a) is representative of what is currently available in **BGL**; it iterates through the vertices, counting

```cpp
using namespace std;
using namespace boost;

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

G g;
//populate g

vector<Vertex> 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))));
```

(a) **BGL**

```cpp
using namespace std;
using namespace graph;

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

using VId = vertex_id_t<G>;

G g;
//populate g

vector<VId> p(size(vertices(g))); //predecessors
vector<int> d(size(vertices(g))); //distances
init_shortest_paths(distance, predecessors);

auto weight_fn =
  [&g](graph::edge_reference_t<graph_type> uv)
     -> int {
        return edge_value(g, uv);
     };









dijkstra_shortest_paths(g, 0, d, p, weight_fn);
```

(b) **graph-v2**

Figure 3: Single Source Shortest Paths Syntax Comparison

the number of triangles incident on every vertex, and adjusts for overcounting at the end.

graph-v2 provides a much more efficient implementation with a high level interface shown in Figure 4(b). The underlying **graph-v2** implementation performs a set intersection of the neighbor list of vertices u and v, only if v is a neighbor of u. This approach requires the edges of a vertex to be stored in lexicographic order (by target vertex id), and to only contain successor edges (target vertex id greater than source vertex id). The latter requirement is equivalent to the graph only containing the upper triangular portion of the adjacency matrix. Then the set intersection is limited to neighbors with vertex ids greater than u and v, avoiding duplicate counting.

In fairness to **BGL**, especially for the purposes of the performance comparison in Section 5, we implement TC in **BGL** using the same set intersection approach used inside **graph-v2**. Figure 5 compares the underlying implementation syntax for each library. Note again for **BGL** the need to go through vertex descriptors to access the out edges of a vertex while **graph-v2** uses a vertex id. [Kevin: Say something about `incidence_iterator` and `end(edges(g,uid))` ? The while condition is different, think this has to be the case, not fully sure why `i != ie` doesn't work for BGL, say something?]

```cpp
using namespace boost;

using G =
  compressed_sparse_row_graph<
     directedS, no_property, no_property>;
using Vertex = graph_traits<G>::vertex_descriptor;

G g;
//populate g

size_t count{0};
for(size_t i = 0; i < N; i++) {
  Vertex cur = vertex(i, g);
  count += num_triangles_on_vertex(g, cur);
}
count /= 6;
```

(a) **BGL**

```cpp
using namespace graph;

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


G g;
//populate g






size_t count = triangle_count(g);
```

(b) **graph-v2**

Figure 4: Triangle Counting Syntax Comparison

```cpp
using namespace boost;

using G =
  compressed_sparse_row_graph<
     directedS, no_property, no_property>;

using edge_iterator = graph_traits<G>::
     out_edge_iterator;

size_t N(num_vertices(g));
size_t triangles(0);

for (size_t uid = 0; uid < N; ++uid) {
  Vertex u = vertex(uid, g);
  std::pair<edge_iterator, edge_iterator>
    u_neighbors = out_edges(u, g);

  auto i = u_neighbors.first;
  auto ie = u_neighbors.second;
  while (i < ie) {
    size_t vid = target(*i, g);
    Vertex v = vertex(vid, g);
    std::pair<edge_iterator, edge_iterator>
      v_neighbors = out_edges(v, g);

    auto i2 = i;
    auto j = v_neighbors.first;
    auto je = v_neighbors.second;

    while (i2 < ie && j < je) {
      size_t wid1 = target(*i2, g);
      size_t wid2 = target(*j, g);
      if (wid1 < wid2) {
        ++i2;
      } else if (wid2 < wid1) {
        ++j;
      } else {
        ++triangles;
        ++i2;
        ++j;
      }
    }
    ++i;
  }
}
```

(a) **BGL**

```cpp
using namespace graph;

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

size_t N(size(vertices(g)));
size_t triangles(0);

for (vertex_id_t<G> uid = 0; uid < N; ++uid) {

  incidence_iterator<G> i(g, uid);
  auto ie = end(edges(g, uid));
  while (i != ie) {
    auto&& [vid, uv] = *i;

    incidence_iterator<G> j(g, vid);
    auto i2 = i;
    auto je = end(edges(g, vid));

    while (i2 != ie && j != je) {
      auto&& [wid1, uw] = *i2;
      auto&& [wid2, vw] = *j;
      if (wid1 < wid2) {
        ++i2;
      } else if (wid2 < wid1) {
        ++j;
      } else {
        ++triangles;
        ++i2;
        ++j;
      }
    }
    ++i;
  }
}
```

(b) **graph-v2**

Figure 5: Triangle Counting Underlying Implementation Syntax Comparison

5 Performance Comparison

5.1 Experimental Setup

To evaluate the performance of this proposed library, we compare its reference implementation (**graph-v2**) against **BGL** and **NWGraph** on a subset of the GAP Benchmark Suite [5]. This comparison includes four of the five GAP algorithms that are in the tier 1 algorithm list of this proposal: breadth-first search (BFS), connected components (CC), single-source shortest paths (SSSP), and triangle counting (TC). The performance of **NWGraph** on the algorithms and a comparison to other graph frameworks was carried out in [6]. Table 3 summarizes the graphs specified by the GAP benchmark. These graphs were chosen with a variety of degree distributions and diameters, and to be large (with edge counts into the billions) but still fit on shared memory machines.

We compare to **BGL** because it the commonly used sequential C++ graph library as described above. **NWGraph** is the direct predecessor of **graph-v2**, with many of the **NWGraph** authors contributing to this library proposal and the **graph-v2** reference implementation. It was implemented with many of the ideas of this proposal in mind, e.g. graphs as a range of ranges and generic algorithms that support any data structure that meet the concept requirements. Since the two implementations are based on similar ideas, we expect similar experimental performance, and include **NWGraph** to verify **graph-v2** does not introduce any performance overhead.

Name	Description	#Vertices (M)	#Edges (M)	Degree Distribution	(Un)directed	References
road	USA road network	23.9	57.7	bounded	undirected	[7]
Twitter	Twitter follower links	61.6	1,468.4	power	directed	[8]
web	Web crawl of .sk domain	50.6	1,930.3	power	directed	[9]
kron	Synthetic graph	134.2	2,111.6	power	undirected	[10]
urand	Uniform random graph	134.2	2,147.5	normal	undirected	[11]

Table 3: Summary of GAP Benchmark Graphs

The **NWGraph** authors published a similar comparison to BGL in which they demonstrated performance improvement of **NWGraph** over BGL [2]. To simplify experimental setup, we rerun these new experiments using the same machine used in that paper, (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). All three implementations were compiled into a single experimental driver to ensure uniform compiler setup (gcc 13.2 using `-Ofast -march=native` compilation flags.) Additionally any graph preprocessing such as symmetricization (for undirected algorithms) or vertex relabeling are guaranteed to be the same for all three implementations.

5.2 Experimental Analysis

Table 4 summarizes our GAP benchmark results for **graph-v2** compared to **BGL** and **NWGraph**. In addition to runtime, the table contains the number of connected components and the number of triangles for each graph as this is helpful for understanding performance. The below subsections consider each GAP algorithm, describe the specific algorithm implementation(s) tested for each library, and examine the performance results.

5.2.1 Breadth-First Search

All implementations of BFS use a sequential push variant that one could find in a textbook (no direction optimization or parallel processing of frontier). As mentioned in Section 4, **BGL** contains support for visitors which is not available in **NWGraph** or the version of **graph-v2** being tested here.

BFS results are competitive between the libraries, with the **graph-v2** implementation achieving the fastest time on all but the road graph. **NWGraph** has noticably worse performance on kron and urand. **BGL** underperforms

Algorithm	Library	Variant	road	twitter	kron	web	urand
BFS	**BGL**		0.99s	7.82s	17.40s	4.13s	59.05s
	NWGraph		0.88s	9.08s	25.04s	2.09s	68.18s
	graph-v2		0.92s	7.00s	15.93s	2.61s	55.13s
CC			1 CC	19.9M CC	71.2M CC	123 CC	1 CC
	BGL	DFS-based	1.30s	32.03s	71.38s	11.93s	94.80s
	graph-v2	DFS-based	0.76s	27.87s	41.21s	6.64s	64.87s
	NWGraph	Afforest	1.15s	6.09s	28.42s	3.29s	28.73s
	graph-v2	Afforest	0.97s	5.85s	23.37s	3.16s	33.84s
SSSP	**BGL**	Dijkstra	3.97s	45.24s	OOM	24.86s	OOM
	NWGraph	Dijkstra	3.62s	95.78s	313.96s	30.66s	356.11s
	graph-v2	Dijkstra	4.06s	104.38s	348.72s	33.77s	387.75s
	NWGraph	DeltaStepping	1.49s	24.48s	74.43s	12.53s	103.97s
TC			439K T	34.8B T	107B T	84.9B T	5.38K T
	BGL	$\frac{1}{6}tr(A^3)$	1.34s	>24H	>24H	>24H	4425.54s
	BGL	Upper triangular	0.61s	1672.71s	8346.70s	251.78s	405.37s
	NWGraph	Upper triangular	0.20s	567.97s	2962.32s	107.85s	152.52s
	graph-v2	Upper triangular	0.17s	524.68s	2683.41s	71.10s	128.32s

Table 4: GAP Benchmark Performance: Time for GAP benchmark algorithms is shown for **BGL**, **NWGraph**, **graph-v2**

on web by 2x but this run only takes around 4s.

5.2.2 Connected Components

The **NWGraph** implementation of CC is based on the Afforest [12] algorithm. **BGL** does not provide an Afforest variant. Instead, **BGL** implements a simple depth-first search based CC algorithm. **graph-v2** contains implementations of both. However, the **graph-v2** implementation of Afforest does not contain support for parallel execution policies which **NWGraph** does, and does not contain the overhead of atomics.

It is likely that other researchers implementing the GAP benchmark use CC to refer to weakly connected components of a directed graph. As the DFS based CC implmentation of **BGL** and **graph-v2** assumes an undirected graph, we make all graphs undirected before running these experiments.

Comparing the two DFS based implementations, **graph-v2** has consistently better performance, up to 2x, over the **BGL** implementation. The Afforest implementations outperform the DFS based implementations. Of the two Afforest implementations, **graph-v2** is slightly faster but this is reasonable considering it does not have the parallel overhead of the **NWGraph** implementation.

5.2.3 Single Source Shortest Paths

For this SSSP comparison, we test a Dijkstra implementation for each of the libraries. **NWGraph** contains multiple Dijkstra implementations, but we use the simplest one which is taken directly from the **NWGraph** benchmark directory. Even though **NWGraph** contains an implementation of Dijkstra, the SSSP results in [2] were based on delta-stepping. In fairness to **NWGraph**, we also include delta-stepping performance for **NWGraph** as this was highly tuned compared to **NWGraph**'s Dijkstra implementation. However, this implementation contains `std::for_each` and is therefore not sequential.

The GAP specification for SSSP only requires that the algorithm compute the shortest distance to every vertex, not the shortest path. Each library has a variant of Dijkstra that only computes shortest distances which we use for these experiments.

SSSP results are more mixed, with superior performance for **BGL** on twitter and web, while **BGL** fails by running out of memory on kron and urand. The edge distances required for SSSP make this a more memory intensive algorithm than the other GAP algorithms. The 2x performance of **BGL** over **NWGraph** and **graph-v2** on twitter is notable and calls for further investigation. [KEVIN: See if we can get kron numbers for **BGL** by doing more memory cleanup]

5.2.4 Triangle Counting

NWGraph and **graph-v2** contain similar implementations of TC that perform a set intersection of the neighbor list of vertices. This is discussed in Section 4 and the **graph-v2** code is shown in Figure 5(b). As noted in Section 4, the naïve **BGL** TC implementation shown in Figure 4(a) is very inefficient. For these performance experiments we include both the inefficient **BGL** approach, and our own **BGL** set intersection implementation shown in Figure 5(b).

TC performance from our naïve **BGL** implementation is far slower than the adjacency matrix set intersection used by **NWGraph** and **graph-v2**. Since the same triangle is counted six times in **BGL**, one can expect at least that much of a slowdown; however, the slowdown is often much worse likely due to poor memory access patterns. The **BGL** implementation of the set intersection approach is much faster than the naïve approach, but is still significantly slower than the **NWGraph** or **graph-v2** implementations, up to a factor of 3x on road and kron. It is unclear if this is a fundamental limitation of **BGL** or our implementation could be further optimized. **graph-v2** consistently outperforms **NWGraph**, up to 1.5x on web. This is surprising given the similarity of the implementations, and could indicate more efficient data access for the **graph-v2** graph data structure.

6 Memory Allocation

Unlike existing STL algorithms, the graph algorithms in the **graph-v2** reference implementation often need to allocate their own temporary data structures. Table 5 records the internal memory allocations required for **graph-v2**'s implementation of the GAP Benchmark algorithms where relevant. It is important to note that the memory usage is not prescribed by the algorithm interface in P3128, and is ultimately determined by the library implementer. Some memory use, such as the queues in BFS and SSSP, will probably be common to most implementations. However, the color map in BFS and the reindex map in CC (used to ensure the resulting component indices are contiguous) could potentially be avoided.

Algorithm	Required Internal Data	Max Size
BFS	queue color map	$O(\|V\|)$ V
CC	reindex map	$O(\|components\|)$
SSSP	priority queue	$O(\|E\|)$
TC	None	NA

Table 5: Internal Memory Allocations of GAP Benchmark Algorithm Implementations in **graph-v2**

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.

References

[1] A. Lumsdaine, K. Deweese, S. McMillan, and P. Ratzloff, "Standard graph library reference implementation, version 2." "https://github.com/stdgraph/graph-v2".

[2] A. Lumsdaine, L. D'Alessandro, K. Deweese, J. Firoz, T. Liu, S. McMillan, P. Ratzloff, and M. Zalewski, "Nwgraph: A library of generic graph algorithms and data structures in c++20." "https://drops.dagstuhl.de/opus/volltexte/2022/16259/".

[3] A. Lumsdaine, L. D'Alessandro, K. Deweese, J. Firoz, T. Liu, S. McMillan, P. Ratzloff, and M. Zalewski, "Nwgraph library code." "https://github.com/pnnl/NWGraph".

[4] J. G. Siek, L.-Q. Lee, and A. Lumsdaine, *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley Professional, Dec. 2001.

[5] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.

[6] A. Azad, M. M. Aznaveh, S. Beamer, M. P. Blanco, J. Chen, L. D'Alessandro, R. Dathathri, T. Davis, K. Deweese, J. Firoz, H. A. Gabb, G. Gill, B. Hegyi, S. Kolodziej, T. M. Low, A. Lumsdaine, T. Manlaibaatar, T. G. Mattson, S. McMillan, R. Peri, K. Pingali, U. Sridhar, G. Szarnyas, Y. Zhang, and Y. Zhang, "Evaluation of graph analytics frameworks using the gap benchmark suite," in *2020 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 216–227, 2020.

[7] "9th DIMACS implementation challenge - Shortest paths.." `http://www.dis.uniroma1.it/challenge9/`, 2006.

[8] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?," *WWW*, 2010.

[9] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," *WWW*, pp. 595–601, 2004.

[10] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the Graph 500," in *Cray User's Group*, CUG, 2010.

[11] P. Erdős and A. Rényi, "On random graphs. I," *Publicationes Mathematicae*, vol. 6, pp. 290–297, 1959.

[12] M. Sutton, T. Ben-Nun, and A. Barak, "Optimizing parallel graph connectivity computation via subgraph sampling," in *IPDPS*, pp. 12–21, IEEE, 2018.