Graph Library: Overview

Document #:	**D3126r3**
Date:	2024-09-12
Project:	Programming Language C++
Audience:	Library Evolution
	SG19 Machine Learning
	SG14 Game, Embedded, Low Latency
	SG6 Numerics
Revises:	D3126r2

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	Soon	**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

D3126r0

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

— Rewrite *Goals and Priorities* section to reflect the structure of the papers and to include a section on our *Future Roadmap*.

— Added *Notes and Considerations* section.

— Concepts will be identified as "For exposition only" until we have consensus of whether they belong in the standard or not.

D3126r1

— Added Issues Status section to be open with the issues that have been reported and that we are working on.

D3126r2

— Add the edgelist as an abstract data structure as a peer to the adjacency list. This completes an open issue for completing the definition of the edgelist.

— Added the `std::graph::edgelist` namespace for edgelist concepts, traits and types to keep identically named types separate from those for adjacency lists.

— Added a reference to the new P3337 *Graph Comparisons* paper to the Getting Started section.

— Update text to make it clear parallel algorithms will *not* be included in the proposal.

D3126r3

— Change the reference implementation from the `std::graph` to the `graph` namespace. This will make it more accessible to the community and allow for easier experimentation outside of this proposal.

— Update the status on supporting more versatile BFS and DFS algorithms.

— Add additional motivation for a graph library in the Overview section.

— Extend the Six Degrees of Kevin Bacon example to include the output and additional description.

— Add a note that we will be unable to support a freestanding graph library in this proposal because of the need for `stack` , `queue` and potential `bad_alloc` exception in many of the algorithms.

— Rename `descriptor` structs to `info` structs in preparation for new BGL-like descriptors.

3 Overview

The original STL revolutionized the way that C++ programmers could apply algorithms to different kinds of containers, by defining *generic* algorithms, realized via function templates. A hierarchy of *iterators* were the mechanism by which algorithms could be made generic with respect to different kinds of containers, Named requirements specified the valid expressions and associated types that algorithms required of their arguments. As of C++20, we now have both ranges and concepts, which provide language-based mechanisms for specifying requirements for generic algorithms.

As powerful as the algorithms in the standard library are, the underlying basis for them is a range (or iterator pair), which inherently can only specify a one-dimensional container. Iterator pairs (equiv. ranges) specify a `begin()` and an `end()` and can move between those two limits in various ways, depending on the type of iterator. As a result, important classes of problems that programmers are regularly faced with use structures that are not one-dimensional containers, and so the standard library algorithms can't be directly used. Multi-dimensional arrays are an example of one such kind of data structure. Matrices do have the nice property that they (typically) have the ability to be "raveled", i.e., the data underlying the matrix can still be treated as a one-dimensional container. Multi-dimensional arrays also have the property that, even though they can be thought of as hierarchical containers, the hierarchy is uniform—an N-dimensional array is a container of N-1 dimensional arrays.

Another important problem domain that does not fit into the category of one-dimensional ranges is that of *graph algorithms and data structures*. Graphs are a powerful abstraction for modeling relationships between entities in a given problem domain, irrespective of what the actual entities are, and irrespective of what the actual relationships are. In that sense, graphs are, by their very nature, generic. Graphs are a fundamental abstraction in computer science, and are ubiquitous in real-world applications.

Any problem concerned with connectivity can be modeled as a graph. Just a small set of examples include Internet routing, circuit partitioning and layout, and finding the best route to take to a destination on map. There are also relationships between entities that are inferred from large sets of data, for example the graph of consumers who have purchased the same product, or who have viewed the same movie. Yet more interesting structures (hypergraphs or k-partite graphs) can arise when we want to model relationships between diverse types of data, such as the graph of consumers, the products they have purchased, and the vendors of the products. And, of course, graphs play a critical role in multiple aspects of machine learning.

Along with these graph abstractions are the graph algorithms that are widely used for solving problems from these domains. Well-known graph algorithms include breadth-first search, Dijkstra's algorithm, connected components, and so on. Because graphs can come from so many different problem domains, they will also be represented with many different kinds of data structures. To make graph algorithms as usable as possible across arbitrary representations requires application of the same principles that were used in the original STL: a collection of related algorithms from a problem domain (in our case, graphs), minimizing the requirements imposed by the algorithms on their arguments, systematically organizing the requirements, and realizing this framework of requirements in the form of concepts.

There are also many uses of graphs that would not be met by a standard set of algorithms. A standardized interface for graphs is eminently useful in such situations as well. In the most basic case, it would provide a well-defined framework for development. But in keeping with the foundational goal of generic programming to enable reuse, it would also empower users to develop and deploy their own reusable graph components. In the best case, such algorithms would be available to the broader C++ programmer community.

Because graphs are so ubiquitous and so important to modern software systems, a standardized library of graph algorithms and data structures would have enormous benefit to the C++ development community. This proposal contains the specification of such a library, developed using the principles above.

In total, all documents, taken as a whole for a Graph Library, propose the addition of **graph algorithms, operators, views, adaptors**, the **graph container interface**, and a **graph container implementation** to the C++ library to support **machine learning** (ML), as well as other applications. ML is a large and growing field, both in the **research community** and **industry**, that has received a great deal of attention in recent years. This documents presents an **interface** of the proposed algorithms, operators, adaptors, views, graph functions, and containers.

4 Goals and Priorities

Because graphs and their algorithms cover a broad range of capabilities and implementations, we have defined a focused set of goals and priorities that will provide an initial set of useful functionality, as well as a sound foundation for future work.

- Provide a firm theoretical foundation for the library.
- Follow the separation of algorithms, ranges, views, and containers established by the standard library.
- Include a rich enough set of algorithms for the library to be useful.
    - The syntax for an algorithm's implementation should be simple, expressive, and easy to understand.
    - The ability to write high-performance algorithms should not be compromised.
    - Algorithms can expect vertices to be in a random access range with an integral vertex id initially.
- Include views for common traversals of a graph's vertices and edges that is concise and consistant without having to use a lower level interface.
    - Simple views for vertexlist, incidence edges on a vertex, neighbors of a vertex, and edges of a graph.
    - Complex views for depth-first search, breath-first search, and topological sort.
- A Graph Container Interface, used by Views and Algorithms, that provides a consistent interface for different graph data structures. The interface includes concepts, types, traits and functions and provides a similar role to the Ranges library for standard containers.
    - Info structs for a consistent data model for vertex, edge and neighbor by views and edge lists.

    - Adjacency list, an outer range of vertices with an inner range of outgoing edges on each vertex.
        - Be able to use the algorithms and views with existing graph data structures using customization points.
        - Support for optional user-defined value types on an edge, vertex, and/or the graph itself.
        - Support bipartite and multipartite graphs.
    - Edgelist, which is a range of edges that allow calling `source_id(e)` and `target_id(e)`, and optionally `edge_value(e)`. This is available for the following.
        - From an `edgelist` view.
        - From a user-defined range of concrete values.
- Provide one or more graph containers that can be used with the algorithms.
    - A high-performance `compressed_graph` container, based on the Compressed Sparse Row matrix.
    - The ability to create simple graph container from standard containers, e.g. `vector<vector<int>>`.

The design should not hinder the ability to extend the functionality to support expanded functionality identified in the future roadmap that follows.

4.1 Future Roadmap

The following are areas we'd like to see in future proposals, after the initial proposals are accepted. We endeavor to investigate these to assure the existing design will support them.

- Additional graph algorithms. The Graph Algorithms paper identifies tiers of algorithms we'd like to see added in the future, including parallel algorithms.
- Support for sparse vertex ids, implying the use of bi-directional containers such as `map` and `unordered_map` for vertices.
- Bi-directional graphs, where vertices have incoming and outgoing edges.
- Constexpr graphs, where vertices and edges are stored in `std::array` or other constexpr-friendly container.
- Parallel graph algorithms.

5 Example: Six Degrees of Kevin Bacon

A classic example of the use of a graph algorithm is the game "The Six Degrees of Kevin Bacon." The game is played by connecting actors to each other through movies they have appeared in together. The goal is to find the smallest number of movies that connect a given actor to Kevin Bacon. That number is called the "Bacon number" of the actor. Kevin Bacon himself has a Bacon number of 0. Since Kevin Bacon appeared with Tom Cruise in "A Few Good Men", Tom Cruise has a Bacon number of 1.

The following program computes the Bacon number for a small selection of actors.

```
std::vector<std::string> actors { "Tom Cruise", "Kevin Bacon", "Hugo Weaving",
                                "Carrie-Anne Moss", "Natalie Portman", "Jack Nicholson",
                                "Kelly McGillis", "Harrison Ford", "Sebastian Stan",
                                "Mila Kunis", "Michelle Pfeiffer", "Keanu Reeves",
                                "Julia Roberts" };

using G = std::vector<std::vector<int>>;
G costar_adjacency_list{
    {1, 5, 6}, {7, 10, 0, 5, 12}, {4, 3, 11}, {2, 11}, {8, 9, 2, 12}, {0, 1}, {7, 0},
    {6, 1, 10}, {4, 9}, {4, 8}, {7, 1}, {2, 3}, {1, 4} };
```

```
int main() {
  std::vector<int> bacon_number(size(actors));

  // 1 -> Kevin Bacon
  for (auto&& [uid,vid] : basic_sourced_edges_bfs(costar_adjacency_list, 1)) {
    bacon_number[vid] = bacon_number[uid] + 1;
  }

  for (int i = 0; i < size(actors); ++i) {
    std::cout << actors[i] << " has Bacon number " << bacon_number[i] << std::endl;
  }
}
```

Output:

```
Tom Cruise has Bacon number 1
Kevin Bacon has Bacon number 0
Hugo Weaving has Bacon number 3
Carrie-Anne Moss has Bacon number 4
Natalie Portman has Bacon number 2
Jack Nicholson has Bacon number 1
Kelly McGillis has Bacon number 2
Harrison Ford has Bacon number 1
Sebastian Stan has Bacon number 3
Mila Kunis has Bacon number 3
Michelle Pfeiffer has Bacon number 1
Keanu Reeves has Bacon number 4
Julia Roberts has Bacon number 1
```

In graph parlance, we are creating a graph where the vertices are actors and the edges are movies. The number of movies that connect an actor to Kevin Bacon is the shortest path in the graph from Kevin Bacon to that actor. In the example above, we compute shortest paths from Kevin Bacon to all other actors and print the results. Note, however, that actor-actor relationships are not how data about actors is available in the wild (from IMDB, for example). Rather, two available types of data are actor-movie and movie-actor relationships. See Section **??** below.

6 What this proposal is not

The Graph Library proposal limits itself to adjacency graphs and edgelists only. An adjacency graph is an outer range of vertices with an inner range of outgoing edges on each vertex. An edgelist is a view of edges on an adjacency list, or a range of edge types.

Parallel graph algorithms are not included in this proposal for several reasons.

— Parallelism is not beneficial for some algorithms, such as for depth-first search.

— There is no clear industry standard for a parallel version of some algorithms.

— The parallel algorithm is a different algorithm altogether, such as *Delta-Stepping* for shortest paths. Omitting them helps to limit the size of this proposal that is already large.

— A richer set of parallelization mechanisms is required because of the irregular and hierarchical nature of graph data structures. Deferring this to a future proposal constrains the complexity and size of this initial proposal.

We feel that providing a broader set of algorithms to address different interests is the better choice. We anticipate that proposals will be submitted for parallel graph algorithms in the future.

Hypergraphs are not supported.

7 Impact on the Standard

This proposal is a pure **library** extension.

8 Interaction wtih Other Papers

Other than the papers identified as part of the Graph Libary, there is no interaction with other proposals to the standard.

9 Implementation Experience

The github github.com/stdgraph repository contains an implementation for this proposal.

10 Usage Experience

There is no current use of the library. There are plans to begin using it in 2024 in a commercial setting.

11 Deployment Experience

There is no current deployment experience of the library. There are plans for this to follow the usage experience.

12 Performance Considerations

The algorithms are being ported from NWGraph to the github.com/stdgraph implementation used for this proposal. Performance analysis from those algorithms can be found in the peer-reviewed papers for NWGraph [1, 2].

13 Prior Art

boost::graph has been an important C++ graph implementation since 2001. It was developed with the goal of providing a modern (at the time) generic library that addressed all the needs of a graph library user. It is still a viable library used today, attesting to the value it brings.

However, boost::graph was written using C++98 in an "expert-friendly" style, adding many abstractions and using sophisticated tempate metaprogramming, making it difficult to use by a casual developer.

NWGraph ([3] and [1]) was published in 2022 by Lumsdaine et al, bringing additional experience gained since creating boost::graph, to create a modern graph library using C++20 for its implementation that was more accessible to the average developer.

While NWGraph made important strides to introduce the idea of the graph as a range-of-ranges and implemented many important algorithms, there are some areas it didn't address that come a practical use in the field. For instance, it didn't have a well-defined API for graph data structures that could be applied to existing graphs, and there wasn't a uniform approach to properties.

This proposal takes the best of NWGraph, with previous work done for P1709 to define a Graph Container Interface, to provide a library that embraces performance, ease-of-use, and the ability to use the algorithms and views on externally defined graph containers.

GraphBLAS Graph algorithms are traditionally developed, and then implemented, using explicit loops over a graph data structure—sometimes referred to as "pointer chasing." An alternative formulation of graph algorithms leverages the close inherent relationship between graphs and sparse matrices to formulate graph algorithms as sequences of higher-level operations: sparse matrix multiplication (and other similar operations) over a semiring [4].

The GraphBLAS is an ad-hoc standardization effort to develop a set of kernel operations for supporting classical graph algorithms. As an API specification, the GraphBLAS is not a a graph library per se, but rather is intended to be used to implement graph algorithms (much as the linear algebra BLAS are used to implement linear algebra libraries such as LAPACK).

A C language binding that specifically implements the API is available as part of SuiteSparse. However, the resulting library relies on its own (opaque) data structures for representing graphs and would not be inter-operable with modern C++ approaches to library and application design. There have been early attempts at native C++ realizations of GraphBLAS, e.g., the GraphBLAS Template Library (GBTL).

(NB: Andrew is a co-author of boost::graph; Scott and Andrew were participants in GraphBLAS standardization and co-authors of GBTL; Andrew, Scott, and Phil are co-authors of NWGraph.)

14 Alternatives

Although the prior efforts have served, and do serve, important roles, they do not meet the needs or expectations of modern C++ development. We are currently unaware of any existing graph library that meets the same requirements and uses concepts and ranges from C++20.

15 Feature Test Macro

The `__cpp_lib_graph` feature test macro is recommended to represent all features in this proposal including algorithms, views, concepts, traits, types, functions, and graph container(s).

16 Freestanding

We are unable to support freestanding implementations in this proposal because many of the algorithms and views require a `stack` or `queue` , which are not available in a freestanding environment. Additionally, `stack` and `queue` require memory allocation which could throw a `bad_alloc` exception.

17 Namespaces

Graph containers and their views and algorithms are not interchangeable with existing containers and algorithms. Additionally, there are some domain-specific terms that may clash with existing or future names, such as `degree` and `partition_id` . For these reasons, we recommend their own namespaces. The following assumption is used in this proposal.

`std::graph` , `std::graph::views` and `std::graph::edgelist`

Alternative locations include the following:

`std::ranges` , `std::ranges::views` , and `std::ranges::edgelist`

`std::ranges/graph` , `std::ranges::graph::views` and `std::ranges::graph::edgelist`

The advantage of these two options are that there would be no requirement to use the ranges:: prefix for things in the std::ranges namespace, a common occurance.

18 Notes and Considerations

There are some interesting observations that can be made about graphs and how they compare and contrast to the standard library that may not be obvious.

— The adjacency list, the primary data structure for this proposal, is a compound data structure of a range of ranges. This introduces a new form of container beyond a simple range.

— There is more than one possible value type, one each for edge, vertex, and graph. Each is optional. This is in contrast to existing practice where the value type is the distinguishing difference between different containers, such as for `set` and `map` .

— Algorithms will often use views, though they can use the GCI functions when needed.

— Algorithms and Views often need to allocate memory internally to achieve their purpose. This is a departure from common practice in the standard.

There are other observations we've also discovered along the way that may not be obvious.

— Storing vertices in a `map` (bi-directional range) requires a different style of programming algorithms, compared to being kept in a `vector` (random access range). When using a `vector` , `edges(g,uid)` would normally be used without much thought. Using that with a `map` would incur a $\mathcal{O}(\log(V))$ cost. Instead, it will use vertex id once to get the vertex reference and then use `edges(g,uv)` . This is expected to result in overloading of existing algorithms based on the range type of a container, distinguished with concepts.

The addition of concepts to the standard library is a serious consideration because, once added, they cannot be removed. We believe that graphs as a range-of-ranges merits the addition new concepts but we recognize that it may be a controversial decision. Toward that end, we will continue to include them to help clarify the examples given and are assumed to be "For exposition only" as suggested implementation until a clear decision to include them, or not, is made.

19 Issues Status

This sections lists the known and open issues for the Graph Library proposal across all papers. They are organized by the paper they are associated with.

19.1 Open Design Issues

— **Pxxxx: Graph Operators** (paper not yet submitted)

1. Complete the paper for additional utility functions including degree, sort, relabel, transpose and join.

— Build on `mdspan` and try to standardize (or at least understand) what might reasonably be called an unstructured span

  — The statement assumes vertices are in a random-access range and prevents the use of bi-directional ranges like std::map, which could be used for sparse vertex ids. The existing design should be able to adept easily to `mdspan`.

  — I don't think I expressed myself very well here. I completely take your point about not assuming that vertices are in a random-access range. But what I'm trying to get at is as follows.

  Suppose someone standardizes unstructured span, as a natural extension of mdspan. What could we learn from its api that may be relevant for graphs? In both cases, we will presumably have a method which allows iteration over the ith partition (or edges of a given node, for graphs). Consistency of the stl may mean we want these to have the same look/feel.

19.2 Open Reported Issues

— P3127 Background and Terminology

1. P1709 has lots of details which I think to be irrelevant. (P1709 is the original proposal that was split into multiple papers)

   — Clarification: I don't find the discussion about adjacency matrices helpful, but rather a distraction. It's not that it shouldn't be there in some form, but at the moment it has a prominence which I don't think is commensurate with its importance to the paper, perhaps exacerbated by the fact that the paper lacks many salient details (see next point).

2. It is very hard to follow
   - Clarification: As it stands, the paper lacks a discussion of the authors' standpoint on graph terminology, defining features (e.g. self loops, multi-edges) and the sort of trade-offs you get by allowing/not allowing them. Put another way, I think the paper would be easier to follow if there's a technical narrative that reveals the way the authors are thinking about this huge area.

     I like the style of the motivation in P1709R5; if this could be greatly extended to include the mathematical background that Andrew is working on, this would be really helpful. And beyond the mathematical background, as discussion of the computational tradeoffs for both graph implementations and the associated algorithms, given certain choice, would be great to have.
   - This paper includes much of the content from P1709R5 for motivation. Andrew will be extending the paper to include a more rigorous mathematical description.
3. We need to add a mathematical perspective to the paper.
   - P3127 includes some of this. We plan on extending it to include a more rigorous mathematical description.
4. There needs to be a proper discussion about whether the paper's definition of graph is what some authors call a multigraph and whether it does/doesn't include loops.
   - The current version of P3128 Algorithms has a summary table for each algorithm that includes Complexity, Directed?, Multi-edge?, Cycles?, Self-loops?, and Throws?. We still need to make a pass through the algorithms to assure the values are correct.
   - The summary tables for the algorithms are necessary but not sufficient:
     - There needs to be a discussion of these aspects for graph implementations themselves. Various graph operations may be more efficient if the graph structure is more constrained. However, not allowing e.g. multiple edges between pairs of nodes prohibits representing many useful systems. There are trade-offs and these need to be discussed.
     - A justification of the choices made for the algorithms may be helpful.
5. The electrical circuit example has issues in P3127, section 6.1.
   - We acknowledge this and will remove it, or replace it with a better example.
   - I think it's very valuable to include electrical circuits in addition to a simpler example. As we've discussed, electrical circuits are surprisingly subtle to represent using graphs, but I think users of a graph library should rightly expect that it can be elegantly done. I think signs of a good design for std::graph is that people can do this. So I think electrical circuits should stay in, in all their glory, but complemented by something less subtle.

- P3128 Graph Algorithms
  1. A concern is that the DFS and BFS functionality isn't flexible enough, especially when compared to boost::graph's visitors.
     - We agree having a more general and flexible BFS and DFS would be valuable. We have investigated the use of coroutines and boost::graph-like visitors and found that the coroutine implementation has better syntax but is 20% slower than the visitor implementation. That kind of performance penalty is not acceptable, making the visitor implementation the clear choice.
     - Visitors have been added to Dijkstra's and Belman-Ford shortest paths algorithms and will be added to BFS, DFS and Topological Sort algorithms in the next revision of P3128.
     - We are also considering adding similar functionality to the views in P3129.
- **P3337: Comparison to boost::graph** (paper not yet submitted)

1. My comment about the structure of the paper changing was a reference to previous comparisons with boost::graph. I'm sure these were in an earlier version, or am I misremembering?
   - We never had any comparisons to boost::graph.
   - We are planning on adding a new paper to compare it to graph-v2 in regards to syntax and performance.

19.3 Resolved Issues

- P3126 Overview
  1. GraphBLAS is not included as part of the prior art.
     - Added in P3126r1.
- P3130 Graph Container Interface
  1. I'm not convinced by the load API.
     - We agree because the use of both load functions and constructors creates ambiguity and complexity when both are defined. Even though constructors weren't in the paper it wasn't clear whether they should be included or not. We have removed the load functions and added constructors for `compressed_graph` to simplify the interface.
  2. Complete the definition of the edgelist concepts, types and CPO functions. This is distinct from the existing edgelist view.

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, 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/".

[2] 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.

[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. Kepner and J. R. Gilbert, eds., *Graph Algorithms in the Language of Linear Algebra*, vol. 22 of *Software, environments, tools*. SIAM, 2011.

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