

Graph Library: Overview

Document #: **P3126r4**
Date: 2026-07-06
Project: Programming Language C++
Audience: Library Evolution
SG19 Machine Learning
SG14 Game, Embedded, Low Latency
Revises: P3126r3

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 <code>compressed_graph</code> 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 how 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

P3126r0

- 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.

P3126r1

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

P3126r2

- 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.

P3126r3

- 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.
- Update the Open Design Issues and Open Reported Issues sections to reflect the current status of the proposal. The changes revolve around the introduction of the new `boost::graph`-like descriptors and improvements to the BFS, DFS and Topological Sort algorithms.

P3126r4

Update the wording to reflect the current status of the proposal along with minor corrections.

- Changed C++ version requirements from C++20 to C++23 throughout all documents.
- Sparse vertex ids and bi-directional graphs are now supported and are no longer future work.
- Remove completed items from the Issues Status section.

3 Overview

The C++ Standard Library (nee the Standard Template Library, or STL), is a well-structured collection of generic software components, systematically organized based on fundamental properties of their underlying problem domain. In the case of the Standard Library, this domain comprises that of one-dimensional containers, specified by iterator pairs (equiv. ranges). Of course, the contents of any one-dimensional container can be another container, to the Standard Library, those contents are opaque.

Since the introduction of the STL and the generic programming paradigm that is its intellectual foundation, there has been recognition of the need to extend the standard library to support hierarchical containers (containers of containers). (See, for example, Matt Austern’s paper “Segmented Iterators and Hierarchical Algorithms, or WG21/N3700 “Hierarchical Data Structures and Related Concepts for the C++ Standard Library”). C++ does have some notion of multidimensional data structures (multi-dimensional ranges, `std::mdspan`), but they are not containers in the sense of the standard library. Moreover, multi-dimensional arrays are not “interesting” hierarchical containers, in that, even if viewed as a container of containers, the container one level down from a given container is still a multi-dimensional array.

Graphs are an important abstraction in Computer Science and that arise in numerous problem domains. They have attracted significant attention recently with the increasing importance of their use in data science, machine learning, and artificial intelligence. Moreover, graphs represent perhaps the most basic form of an interesting (non-trivial) hierarchical data structure. For example, although the “adjacency list” representation of a graph only comprises two levels, it is richer than a two-dimensional container. Most significantly, the “inner container” of the adjacency list cannot be opaque; traversal of a graph is of necessity an interleaved traversal of the inner and outer containers.

4 Goals and Priorities

The main goal of this library is to provide a self-consistent and systematic library of software components for graph computations, based on well-defined graph representations. It includes generic algorithms, named requirements, views, and utilities. It is informed by the authors' past experience with the Boost Graph Library (BGL), the NWGraph library, the C++ GraphBLAS, proprietary graphs embedded in products, and the library originally proposed in P1709. The proposed library seeks to leverage the existing Standard Library while introducing as little new machinery as possible. Because of the broad scope of graph data structures and algorithms, we have defined a focused set of goals and priorities to deliver an initial set of useful and practical functionality that will also establish a solid foundation for future work.

Design Principles. We propose a lightweight library of software components for graph computing that is self-consistent and systematically defined. The foundation of the library is its collection of generic traversal patterns, views and algorithms. Although concepts will be used “under the hood,” the API for the library’s algorithms and traversals will be specified using named requirements. As little new machinery as possible is introduced; to the greatest extent possible, graph library components are built on top of existing Standard Library components.

Named Requirements. The named requirements in the library reflect the needs of graph algorithms and are organized and named according to well-accepted practice in the underlying problem domain of graphs and graph algorithms. The named requirements are intended to be minimal in order to enable the largest variety of concrete data structures to satisfy them.

Graph Representation. To be precise, and in keeping with common practice, the named requirements, the underlying concepts, and the algorithms in the library are defined in terms of representations of graphs. What we mean by a graph representations is developed in detail in the companion document P3127.

Leverage the Standard Library. With the basis of the library being the representation of graphs in the form of hierarchical containers, and with the intention for the library to be included as part of the C++ standard library, we leverage standard library components to the greatest extent possible. Notably, compositions of Standard Library ranges satisfy the graph representation requirements, sufficient for their use with the proposed graph library.

Graph Library Components. The library will comprise algorithms and views, along with domain-specific named requirements.

[ANDREW: This is still in flux. I imagine a diagram and a table of some kind.] [PHIL: I believe this is done with the concepts diagrams in P3130. Is that sufficient?]

4.1 Future Roadmap

The following areas are opportunities for future proposals, after the initial proposals are accepted. We endeavor to investigate them (without introducing additional proposals) to ensure the currently proposed design will support them.

- Additional graph algorithms. The Graph Algorithms proposal (P3128) identifies tiers of algorithms that we suggest be added in a staged fashion (including parallel algorithms).
- 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 lists and edgelists only. An adjacency list 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; they are indistinguishable in terms of the edges they represent to an algorithm.

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 with Other Papers

The entirety of our proposal for graph algorithms and data structures comprises multiple companion papers: [P3127](#) (Background and Terminology), [P3128](#) (Algorithms), [P3129](#) (Views), [P3130](#) (Container Interface), [P3131](#) (Containers) and [P3337](#) (Comparison to other graph libraries). Other than these papers, there are no interactions with other proposals to the standard.

9 Implementation Experience

The github github.com/stdgraph/graph-v3 repository contains a reference implementation for this proposal.

10 Usage Experience

There is no current use of the library outside of the proposers. We are working on plans to make it available for broader use in commercial, academic, and research settings.

11 Deployment Experience

There is no current deployment experience of the library. Deployment experience will be gathered in conjunction with use.

12 Performance Considerations

The algorithms are being ported from Boost Graph and NWGraph to the github.com/stdgraph/graph-v3 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

[PHIL: Summarize issues with each library WRT standardization as a basis. Demonstrate issues in P3337 Comparison.]

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 template metaprogramming, making it difficult to use by a casual developer. Particular pain-points described in ad-hoc discussions with users include: property maps, parameter-passing, and adapting to existing graph data structures.

This library draws on the experience gained from **boost::graph** to provide a more modern and accessible C++ graph library. The reference library has been extended to support many of the features and capabilities of **boost::graph**.

(NB: Andrew is a co-author of `boost::graph`)

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++23 for its implementation that was more accessible to the average developer.

While NWGraph made important strides to introduce the idea of an adjacency list as a range-of-ranges and implemented many important algorithms, there are some areas it didn't address that are of 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 adjacency lists defined with standard containers.

(Scott and Andrew were participants in GraphBLAS standardization and co-authors of GBTL.)

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 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).

(Andrew, Scott, and Phil are co-authors of NWGraph.)

[PHIL: [Add discussion of LEMON library.](#)]

[PHIL: [Add discussion of NetworkX library.](#)]

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++23.

[PHIL: [Comment on other graph libraries in github](#)]

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 Language Requirements

The library targets C++23 as its baseline.

The reference implementation provides backward compatibility to C++20 via an external `expected` library (e.g., `tl::expected`). Switching it to require C++23 may limit those who can use it, so care will be taken when deciding to require C++23.

18 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 separate namespace(s) for the graph functionality. The following assumption is used in this proposal.

`std::graph::adj_list`, `std::graph::views` and `std::graph::edge_list`

Alternative locations include the following:

`std::ranges`, `std::ranges::adj_list`, `std::ranges::edge_list`, and `std::ranges::views`

`std::ranges::graph`, `std::ranges::graph::adj_list`, `std::ranges::graph::edge_list`, `std::ranges::graph::views`

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 occurrence.

19 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`. This decision was made to avoid the combinatorial explosion of types that would otherwise occur if each combination of edge, vertex, and graph value types required a separate container type.
- Algorithms will often use views, though they can use the GCI functions when needed.
- Algorithms and Views often need to allocate memory internally because of the traversal requirement of using a stack or queue. This is a departure from common practice in the standard.

The addition of concepts to the standard library is a serious consideration because, once added, they cannot be removed. We believe that adjacency lists as a range-of-ranges merits the addition of 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.

20 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.

20.1 Open Design Issues

20.2 Open Reported Issues

- [P3128 Graph Algorithms](#)

1. 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.
 - Improvements to the Dijkstra and Bellman-Ford algorithms have been made. Please review give feedback.
 - A justification of the choices made for the algorithms may be helpful.
- [P3337 Comparison to Other Graph Libraries](#)
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`.
 - The paper was published 2025-07-30. A new version with updated benchmark numbers is in process with descriptors functionality.
 - Additional comparison with other graph libraries beyond `boost::graph` is needed, e.g. LEMON, igraph, NetworkX, etc. More content is also needed to describe why they aren't suitable as a basis for standardization.

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., ISO CPP 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. Aznavah, 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.