


NWGraph: A Library of Generic Graph Algorithms and Data Structures in C++20


Andrew Lumsdaine ✉ 
University of Washington, Seattle, WA, USA
Pacific Northwest National Laboratory,
Richland, WA, USA
TileDB, Inc., Cambridge, MA, USA

Luke D'Alessandro ✉
Indiana University, Bloomington, IN, USA

Kevin Deweese ✉
Cadence Design Systems, San Jose, CA, USA

Jesun Firoz ✉ 
Pacific Northwest National Laboratory,
Richland, WA, USA

Xu Tony Liu ✉ 
University of Washington, Seattle, WA, USA

Scott McMillan ✉ 
Software Engineering Institute,
Carnegie Mellon University, Pittsburgh, PA, USA

John Phillip Ratzloff ✉
SAS Institute, Cary, NC, USA

Marcin Zalewski ✉
NVIDIA, Seattle, WA, USA

Abstract

The C++ Standard Library is a valuable collection of generic algorithms and data structures that improves the usability and reliability of C++ software. Graph algorithms and data structures are notably absent from the standard library, and previous attempts to fill this gap have not gained widespread adoption. In this paper we show that the richness of graph algorithms and data structures can in fact be captured by straightforward composition of existing C++ mechanisms. Generic programming is algorithm-oriented. Accordingly, we apply a systematic approach to analyzing a broad set of graph algorithms, “lift” unnecessary constraints from them, and organize the resulting set of minimal common *type requirements*, i.e., concepts, for defining their interfaces. By using the newly available *ranges* and *concepts* in C++20, the type requirements for generic graph algorithms can be succinctly expressed. The generic algorithms and data structures resulting from our analysis are realized in NWGraph, a modern, composable, and extensible C++ library.

2012 ACM Subject Classification Software and its engineering → Software libraries and repositories; Mathematics of computing → Graph algorithms

Keywords and phrases Graph library, generic programming, graph algorithms

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.31

Supplementary Material *Software (Source Code)*: <https://github.com/pnnl/NWGraph>
archived at `swh:1:dir:50db7d4a73652c1073d8141a1e3d83896b0ca3b0`

Funding This work was partially supported by the High Performance Data Analytics (HPDA) program and 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. This research was partially supported by NSF Awards OAC-1716828 and OAC-2126266. This material is also based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center [DM22-0432].

1 Introduction

Graphs are powerful mathematical tools for reasoning about the relationships between given entities, focusing on the *structure and characteristics of the relationships*, independent of what the entities and the relationships actually are. Consequently, results from graph theory



© Andrew Lumsdaine, Luke D'Alessandro, Kevin Deweese, Jesun Firoz, Xu Tony Liu, Scott McMillan, John Phillip Ratzloff, Marcin Zalewski, and Carnegie Mellon University;
licensed under Creative Commons License CC-BY 4.0

36th European Conference on Object-Oriented Programming (ECOOP 2022).

Editors: Karim Ali and Jan Vitek; Article No. 31; pp. 31:1–31:28



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

can be applied to any actual sets of data for which relationships between elements can be established. Internet packet routing, molecular biology, electronic design automation, social network analysis, and search engines are just some of the problem areas where graph theory is regularly applied. The general applicability we find in graph theory – the *genericity*, if you will – is a goal for software libraries as well as mathematical theories; graph algorithms and data structures (collectively, “graphs”) would seem to be ideally suited for software reuse.

Realizing a truly generic library for graphs has significant challenges in practice. Graphs in theory are useful because they are abstract, but, in practice, they have to be made concrete when used to solve an actual problem. That is, graphs in practice do not manifest themselves in the abstract form to which theory and abstract algorithms are applied. Rather, they are often encoded in some domain-specific form or are latent in problem-related data. And even if a domain programmer constructs a graph from their data, the domain-specific graph data structure might not be compatible with the API of a given graph library.

The celebrated Standard Template Library (STL), now part of the C++ standard library, addressed this problem for fundamental algorithms and abstract containers of data elements [30]. With the STL, *generic programming* emerged as a software-development sub-discipline that focused on creating frameworks of reusable and composable libraries. Fundamental to the philosophy of generic programming is that algorithms should be able to be composed with arbitrary *types*, notably types that may have been developed completely independently of the library. To achieve this goal, generic algorithms are specified and written in terms of *abstract properties of types*; a generic algorithm can be composed with any type meeting the properties that it depends on. Philosophically, generic programming goes hand-in-glove with the abstraction process inherent in graph theory. Graphs are abstract models of entities in relationship – a graph algorithm should be able to operate directly on the entities and relationships in a programmer’s data.

It is not just the philosophy of generic programming from the STL that can be leveraged to develop a generic graph library. In fact, an important principle upon which our work is based is that **the standard library already contains sufficient capability to support graph algorithms and data structures**. The *type requirements for generic graph algorithms* can be expressed using existing type requirement machinery for standard library algorithms, and useful and efficient graph algorithms can be implemented based on these requirements.

We apply this principle to develop NWGraph, a generic library of algorithms for graph computation that are independent of any particular data structure (in particular, independent of any particular graph data structure). Following current generic library practice, NWGraph algorithms are organized around a minimal set of common requirements for their input types (these requirements are formalized in the form of C++20 *concepts*).

The foundation of this paper is a requirements analysis from which we derive a uniform set of type requirements for graph algorithms; those requirements subsequently reified as C++ concepts. Based on this foundation, we construct the primary components of NWGraph: **algorithms**, defined and implemented using our concepts; **adaptors**, for converting one representation of a graph into another and for enabling structured traversals, and **data structures** that model our foundational requirements.

NWGraph contains the following innovations:

- A concept taxonomy (expressed using C++20 concepts) for specifying graph algorithm requirements;
- Characterization of graphs using standard library concepts (as a random access range of forward ranges);
- A rich set of range adaptors for accessing and traversing graphs;
- An API designed to fully support modern idiomatic C++;

- An efficient and fully parallelized implementation, using C++ execution policies and Intel® Threading Building Blocks; and
- Application of the generic programming process to minimize requirements on algorithm input types (thereby enlarging the scope of composability).

In the following sections we first provide some basic background and terminology that we will be using to discuss graph algorithms (§2) as well as a bit more detail on generic programming (§3). Next, we analyze the domain of graph algorithms with respect to common requirements and present the fundamental concepts in NWGraph (§4). We then present an overview of the primary components of NWGraph in addition to its concepts: its algorithms (§5), adaptors (§6), and data structures (§7). We include abstraction penalty experiments, evaluate the performance of our library in comparison with other well-known graph libraries, and conduct a strong scaling study of the parallel performance of NWGraph (§8). Finally, we provide a high-level feature comparison of NWGraph with other extant graph libraries (§9) and conclude with some of our observations and experiences in developing NWGraph (§10). NWGraph is hosted at <https://github.com/pnnl/NWGraph>.

2 Graph Background

We define a *graph* G as comprising two finite sets, $G = \{V, E\}$, where the set $V = \{v_0, v_1, \dots, v_{n-1}\}$ is a set of entities of interest, “vertices” or “nodes,” and $E = \{e_0, e_1, \dots, e_{m-1}\}$ is a set of pairs of entities from V , “edges” or “links.” Edges may be ordered or unordered; a graph defined with ordered edges is said to be *directed*; a graph defined with unordered edges is said to be *undirected*.

► **Remark.** Understanding graphs is necessary to develop requirements for algorithms. However, it should be noted that we don’t derive those requirements from the graph model, but *instead from the algorithms*. This is a key distinction between generic programming and, say, Object-Oriented (OO) requirements analysis.

2.1 Representing Graphs

To define algorithms on graphs and to be able to reason about those algorithms, we need to define some representations for graphs; not much can be done computationally with abstract sets of vertices and edges. The specific characteristics of these representations are what we use to express algorithms (still abstractly) but when those algorithms are implemented as generic library functions, those characteristics will in turn become the basis for the library’s interfaces (represented in our case as C++ concepts).

One of the fundamental operations in graph algorithms is a *traversal*. That is, given a vertex u , we would like to find the *neighbors* of u , i.e., all vertices v such that the edge (u, v) is in the graph. Then, for each of those edges, we would like to find their neighbors, and so on. The representation that we can define to make this efficient is an *adjacency list*.

Given a graph $G = (V, E)$, we can define an adjacency-list representation in the following way. Assign to each element of V a unique index from the range $[0, |V|)$ and denote the vertex identified with index i as $V[i]$. We can now define a new graph with the same structure as G , but in terms of the indices in $[0, |V|)$, rather than with the elements in V . Let the *index graph of G* be the graph $G' = (V', E')$, where $V' = [0, |V|)$ and E' consists of $|E|$ pairs of indices from V , such that a pair (i, j) is in E' if and only if $(V[i], V[j])$ is in E . Which is all to say, the index graph of G is the graph we get by replacing all elements of G with their corresponding indices.

We make the following definition: An *adjacency list* of an index graph $G = (V, E)$ is an array $Adj(G)$ of size $|V|$ (the array is indexed from 0 to $|V| - 1$) with the following properties:

- $Adj(G)$ is a container of $|V|$ containers, one container for each vertex in V , and
- The container $Adj(G)[u]$ contains all vertices v for which there is an edge $(u, v) \in E$.

This structure, an adjacency list of an index graph, or an index adjacency list, is the fundamental structure used by almost all graph algorithms.

► **Remark 1.** Although the standard term for this kind of abstraction is “adjacency list”, and although it is often drawn schematically with linked lists as elements, it is not necessary that this abstraction be implemented as an actual linked list. In fact, other representations (such as compressed sparse row storage) are significantly more efficient, as we show in Section 8.3. What is important is that the items that are stored, vertex indices, can be used to index into the adjacency list to obtain other lists of neighbors.

► **Remark 2.** An adjacency list does not store edges per se, rather it stores lists of reachable neighbors. Therefore, though it can represent a directed or undirected graph, an adjacency list is structurally neither inherently directed nor undirected. That is, given vertex u , the container $Adj(G)[u]$ contains the vertex v if the edge (u, v) is contained in E , i.e., for a directed graph with edge (u, v) in E , $Adj(G)[u]$ will contain v . For an undirected graph with edge (u, v) contained in E , $Adj(G)[u]$ will contain v **and** $Adj(G)[v]$ will contain u . Thus, directedness of the original graph is made manifest in the *values* stored in the adjacency list, not in its structure.

3 Generic Programming

Generic programming is a software development paradigm inspired by the organizational principles of mathematics [31]. That is, a generic library comprises a framework of algorithms in a problem domain, based on a systematic organization of common type requirements for those algorithms. The type requirements themselves, specified as *concepts*, are part of the library as well, and provide the interface that enables composition of library components with other, independently-developed, components. Concrete types that meet the requirements of a concept are said to *model* that concept. As an example, the `iterator` concept taxonomy was the foundation upon which the STL was organized [21,30].

Generic algorithms (that is, algorithms in a generic library) are designed so that the requirements they impose on types are as minimal as possible without compromising efficiency, thus enabling the widest scope of potential composition, and therefore, reuse. Generic algorithms are derived from concrete ones, which are gradually made more generic by removing (“lifting”) unnecessary requirements. This process continues as long as instantiation of the generic algorithm with concrete types remains as efficient as the equivalent concrete algorithm would have been.

3.1 Lifting

The first (and major) phase of the generic programming process is sometimes known as “lifting” where we create generic algorithms through a process of successive generalization. That is, the process is

1. Study the concrete implementation of an algorithm;
2. Lift away unnecessary requirements to produce a more abstract algorithm;
3. Repeat the lifting process until we have obtained a generic algorithm that is as general as possible but that still instantiates to efficient concrete implementations; and
4. Catalog remaining requirements and organize them into concepts.

■ **Listing 1** Concrete implementation for summing elements of an array.

```
int sum(int *array, int n) {
    int s = 0;
    for (int i = 0; i < n; ++i) {
        s = s + array[i]; }
    return s; }
```

■ **Listing 2** Lifted implementation of sum, where traversal through the container and element access has been abstracted through the use of iterators and addition has been further lifted with the introduction of the operator parameter `op`.

```
template <class Iter, class T, class Op>
T accumulate(Iter first, Iter last, T s, Op op) {
    while (first != last) {
        s = op(s, *first++); }
    return s; }
```

Listings 1–2 show two concrete implementations of a `sum` algorithm. The first steps through an array of integers, indexing into the array at each step and summing the resulting value into `s`. Instead of an array, any eligible container (for example, linked list) can store the values.

The authors of the STL realized the commonality of traversal and element access across most basic computer science algorithms. The requirements for traversal and access were generalized and unified into a hierarchy of type requirements for iterators [30].

An iterator-based algorithm for accumulating elements in a container is shown in Listing 2. Note that this single parameterized algorithm replaces the `sum` algorithm shown in Listing 1 (and more). The process of summation has further been generalized by the introduction of a function object `op` as a parameter to the function.

3.2 Specialization

In generic programming, the dual to lifting is *specialization*. That is, once an algorithm is lifted and made generic, it is specialized through composition with a concrete data type to realize a concrete implementation of the algorithm. Listing 3 shows two example usages of the generic `accumulate`, composing it with an array as well as a linked list from the STL.

Now, there is a crucial requirement that is part of specialization. In generic programming, we don't just require that when we have a lifted algorithm that we can compose it with the data types that we lifted from. In addition to that basic requirement, we also require that *there is zero abstraction penalty*. That is, the specialized generic algorithm should provide

■ **Listing 3** Specializations of the generic `accumulate` algorithms shown in Listing 2. The `accumulate` algorithm is composed with an integer array (left) and `accumulate` is composed with a linked list (right).

```
int* array = new int [10];
int result =
    accumulate(array, array + 10,
        0, std::plus<int>());
```

```
std::forward_list<double> ptr;
double result = accumulate(ptr,
    nullptr, 0.0,
    std::times<double>());
```

■ **Listing 4** Skeleton of the requirements for a C++ `input_iterator`.

```

1  template <class I>
2  concept input_iterator = requires(I i) {
3      typename std::iter_value_t<I>;
4      typename std::iter_reference_t<I>;
5      { *i } -> std::same_as<std::iter_reference_t<I>>;
6      { ++i } -> std::same_as<I &>;
7      i++;};

```

exactly the same performance as the concrete algorithm from which it was lifted, when composed with the original types that were lifted. With modern compilers and libraries, this requirement is actually met, and is one of the reasons that libraries such as the C++ standard library have been so successful in practice.

3.3 Concepts in C++20

In generic programming, concepts consist of valid expressions and associated types, which define a family of allowable types admissible for composition with generic algorithms. Introduced as a language feature for C++20, concepts constrain the set of types that can be substituted for class and function template arguments. This development has been instrumental in the notable development of the ranges algorithm library taxonomy, serving as the link between generic algorithm interface and implementation [23].

A C++20 `concept` definition declares a set of requirements on types. There are four types of requirements:

- A simple requirement consists of an arbitrary expression statement. The requirement is that the expression is valid.
- A type requirement consists of the keyword `typename` followed by a type name, optionally qualified. The requirement is that the named type exists.
- A compound requirement specifies a conjunction of arbitrary constraints such as expression constraint, exception constraint, and type constraint, etc.
- A nested requirement consists of another `requires`-clause, terminated with a semicolon. This is used to introduce predicate constraints expressed in terms of other named concepts applied to the local parameters.

Listing 4 shows the skeleton of the C++ `concept` definition for `input_iterator`. As hinted in our example, this concept specifies that an `input_iterator` can be de-referenced with `operator*` (line 5) and incremented with `operator++` (lines 6 and 7). Additionally, the concept specifies two associated types: `std::iter_value_t<I>` and `std::iter_reference_t<I>`. Line 5 also indicates that the expression `*i` returns the same type as `std::iter_reference_t<I>`. Again, this example is abbreviated for purposes of illustration. A complete description of the C++20 standard library concepts (including the iterator hierarchy) can be found online at <https://en.cppreference.com/w/cpp/concepts>.

3.4 Ranges in C++20

The new C++20 Ranges library [23] generalizes iterators and containers in C++. Ranges provide a way to make STL algorithms *composable* and improve the readability and writability of C++ code. Ranges consist of a pair of begin and end iterators, which are not required to be the same type. An example of using `ranges` is:

```
std::vector<int> v { /* ... */ }
std::min_element(v.begin(), v.end()); //iterator API
std::ranges::min_element(v);         //ranges API
```

In the first case, the generic `min_element` function is called with an iterator pair (`begin` and `end` of the container `v`). In the second case, `min_element` function is called directly with `v` as the parameter, as a `std::vector` is a range (specifically, it satisfies the requirements for the `random_access_range` concept).

C++20 ranges are defined in terms of C++20 concepts. A `std::range` itself is a very straightforward concept:

```
template <class T>
concept range = requires(T& t) {
    ranges::begin(t);
    ranges::end(t); };
```

It has two valid expressions: `begin` and `end`. The `std::input_range`, which abstracts containers that have forward iterators, is thus defined:

```
template<class T>
concept input_range = ranges::range<T>
    && std::input_iterator<ranges::iterator_t<T>>;
```

This definition states that an `input_range` is a `range` and that the iterator type associated with that range meets the requirements of the `std::input_iterator` concept.

Related to our development of graph concepts, two range concepts of particular relevance include `ranges::forward_range`, which allows iteration over a collection from beginning to end multiple times (as opposed to an input iterator which is only guaranteed to be able to iterate over a collection once) and `ranges::random_access_range`, which further allows indexing into a collection with `operator[]` in constant time.

4 Generic Graph Algorithms

In this section we analyze the requirements for graph algorithms in order to derive generic graph algorithms. `NWGraph` realizes these generic algorithms as function templates, and realizes the type requirements as C++20 concepts. Our process centers on defining type requirements at the interfaces to algorithms based on what the algorithms actually require of their types, rather than starting with graph types and building algorithms to those types.

4.1 Algorithm Requirements

Algorithms in the STL operate over containers. The concepts defined for the STL have to do with mechanisms for traversing a container and accessing the data therein. Since graphs in some sense are also containers of data, we can reuse the mechanisms from the STL for traversing graphs and accessing graph data, to the extent that makes sense. However, graphs are *structured data* and graph algorithms traverse that structure in various ways. Accordingly, our graph concepts must support structured traversal of graphs.

Most (but not all) graph algorithms traverse a graph vertex to vertex by following the edges that connect vertices. For implementing such algorithms, it is assumed that a graph $G = \{V, E\}$ is represented with an adjacency-list structure¹ as defined in Section 2.1.

¹ The reader is reminded that although the term of art is “adjacency list,” containers other than lists can be used to store neighbor information.

<pre> BFS(<i>G</i>, <i>s</i>) 1 for each vertex <i>u</i> ∈ <i>V</i>(<i>G</i>) 2 <i>color</i>[<i>u</i>] ← WHITE 3 <i>color</i>[<i>s</i>] ← GRAY 4 <i>Q</i> ← ∅ 5 ENQUEUE(<i>Q</i>, <i>s</i>) 6 while <i>Q</i> ≠ ∅ 7 <i>u</i> ← DEQUEUE(<i>Q</i>) 8 for each <i>v</i> ∈ <i>Adj</i>(<i>G</i>)[<i>u</i>] 9 if <i>color</i>[<i>v</i>] = WHITE 10 <i>color</i>[<i>v</i>] ← GRAY 11 ENQUEUE(<i>Q</i>, <i>v</i>) 12 <i>color</i>[<i>u</i>] ← BLACK </pre>	<pre> void bfs(const Graph& G, int s) { ... for (int u = 0; u < size(G); ++u) color[u] = WHITE; color[s] = GREY; std::queue<int> Q; Q.push(s); while (!Q.empty()) { auto u = Q.front(); Q.pop(); for (auto&& v : G[u]) { if (color[v] == WHITE) { color[v] == GREY; Q.push(v); }} color[u] = BLACK; }} </pre>
--	---

■ **Figure 1** Pseudocode and C++ implementation of breadth-first search. Existing C++ language mechanisms and library components are expressive enough to essentially realize the algorithm line for line.

4.2 Requirements for Concrete Algorithms

A prototypical algorithm in this class is the breadth-first search (BFS) algorithm. The pseudocode for this algorithm, along with its C++ implementation, is shown in Figure 1. The algorithm is abbreviated from [8]. Modulo some type declarations that would be necessary for real code to compile, but which can be omitted from pseudocode, the C++ code, using out-of-the-box language mechanisms and library components, has essentially a one-one correspondence to the pseudocode.

From this implementation we can extract an initial set of requirements for the BFS algorithm:

- The graph G must meet the requirements of a *random access range*, meaning it can be indexed into with an object (of its difference type) and it has a size.
- The value type of G (the inner range of G) must meet the requirements of a *forward range*, meaning it is something that can be iterated over and have values extracted.
- The value type of the inner range must be something that can be used to index into G .
- All elements stored in G must be able to correctly index into it, meaning their value are between 0 and $\text{size}(G)-1$, inclusive.

Associated with the concepts of a random access range and forward range are complexity guarantees (which are also implied by the theoretical algorithm). Indexing into G is a constant-time operation and iterating over the elements in $G[u]$ is linear in the number of elements stored in $G[u]$.

As an example, we could use any of the following compositions of standard library components for the `Graph` datatype above:

```

using Graph = std::vector<std::list<int>>;
using Graph = std::vector<std::vector<unsigned>>;
using Graph = std::vector<std::forward_list<size_t>>;

```

(In fact, any `Graph` data structure meeting the above requirements could be used.)

We now have a set of requirements for a concrete implementation of BFS. Following the generic programming process, there are various aspects of the implementation that we could begin lifting. Ultimately, as with the STL, we want a set of concepts useful across families

<pre> DIJKSTRA(G, w, s) 1 for each vertex $u \in V(G)$ 2 $d[u] \leftarrow \infty$ 3 $\pi[u] \leftarrow \text{NIL}$ 4 $d[s] \leftarrow 0$ 5 $Q \leftarrow V(G)$ 6 7 while $Q \neq \emptyset$ 8 $u \leftarrow \text{EXTRACT-MIN}(Q)$ 9 for each $v \in \text{Adj}(G)[u]$ 10 if $d[u] + w(u, v) < d[v]$ 11 $d[v] \leftarrow d[u] + w(u, v)$ 12 $\pi[v] = u$ </pre>	<pre> void dijkstra(const Graph& G, int s) {... for (int u = 0; u < size(G); ++u) { d[u] = INF; pi[u] = NIL; } d[s] = 0 for (int u = 0; u < size(G); ++u) Q.push({u, d[u]}); while (!Q.empty()) { auto [u, x] = Q.top(); Q.pop(); for (auto&& [v, w] : G[u]) { if (d[u] + w < d[v]) { d[v] = d[u] + w; pi[v] = u; } } } } </pre>
--	---

■ **Figure 2** Pseudocode and C++ implementation of Dijkstra’s algorithm. As with BFS, existing C++ language mechanisms and library components are expressive enough to essentially realize the algorithm line for line.

of graph algorithms. So rather than lifting BFS in isolation, we now examine concrete implementations of other algorithms in order to identify common functionality that can be lifted in order to unify abstractions.

Figure 2 shows the pseudocode and corresponding C++ implementation for Dijkstra’s algorithm for solving the single-source shortest paths problem. From this implementation we can extract an initial set of requirements for the concrete `dijkstra` algorithm:

- The graph `G` is a *random access range*.
- The value type of `G` (the inner range of `G`) is a *forward range*.
- The value type of the inner range is a pair, consisting of a something we will call a vertex type and something we will call a weight.
- The vertex type is something that can be used to index into `G`.
- All values stored as vertex types in `G` must be able to correctly index into `G` meaning their value are between `0` and `size(G)-1`, inclusive.

Just as the code of `dijkstra` is similar to `bfs`, some of these requirements are also the same. **However, the key difference is in what is stored inside of the graph.** This implementation of `dijkstra` assumes that the graph stores a tuple consisting of a *vertex value and an edge weight*. That is, rather than the `Graph` types shown above, we could use the following for `dijkstra`:

```

using Graph = std::vector<std::list<std::tuple<size_t, int>>>;
using Graph = std::vector<std::vector<std::tuple<unsigned, double>>>;
using Graph = std::vector<std::forward_list<std::tuple<int, float>>>;

```

This is a different kind of graph than we had for `bfs`, which only stored a value. Yet, even a graph that stores a weight on its edge should be suitable for BFS exploration. Similarly, a graph without a weight on its edge should be suitable for Dijkstra’s algorithm, provided a weight value can be provided in some way (or a default value, say, 1, used).

4.3 Lifting

From the foregoing discussion, we have two pieces of functionality we need to lift. First, we need to lift how the neighbor vertex is stored so that whether it is stored as a direct value or as part of a tuple (or any other way), it can be obtained. Second, we need to lift how

weights (or, more generally, **properties**) are stored on edges. And, finally, implied when we say we want to use different kinds of graphs with these algorithms, we need to parameterize them on the graph type (make them function templates rather than functions).

4.3.1 Parameterizing the Graph Type

In this lifting process we will be building up to a concept, which we will illustrate by lifting `dijkstra`. We begin by presenting its type parameterization. The prototype for a `dijkstra` function template based on our previous definition would be

```
template <class Graph>
auto dijkstra(const Graph& G, vertex_id_t<G> s);
```

Note that we have parameterized *two* things: the `Graph` type itself, as well as the type of the starting vertex `s`. In this case, the vertex type is not arbitrary, it is related to the type of the graph, and so we have a type primitive `vertex_id_t` that returns the type of the vertex associated with graph `G`.

- We can update some of the previous requirements for the type-parameterized `dijkstra`:
- `Graph` must meet the requirements of `random_access_range`.
 - The value type of `Graph` (the inner range of `Graph`) must meet the requirements of `forward_range`.
 - The type `vertex_id_t<Graph>` is an associated type of `Graph`.
 - The type `vertex_id_t<Graph>` is convertible to the `range_difference_t` of `Graph` (that is, it can be used to index into a `Graph`).

Both classes of graphs that we had previously seen for `bfs` and `dijkstra` satisfy these requirements (which are more general than either of the previous requirements). For example, both of the following compound structures

```
std::vector<std::vector<int>>;
std::vector<std::vector<std::tuple<int, float, double>>>;
```

satisfy the lifted requirements, (provided a suitable overload of `vertex_id_t` is defined) though each would have only satisfied one of the previous requirements. Note however, we still need to do more lifting before we can compose `bfs` or `dijkstra` with these types.

4.3.2 Lifting Neighbor Access

How a neighbor is stored is dependent on the graph structure itself; the mechanism for accessing it should therefore vary based on the graph type. In keeping with standard C++ practice – and since we want to be able to use C++ standard library containers, we adopt a polymorphic free function interface to abstract the process of accessing a neighbor. In particular, we define a *target customization point object* (CPO) to abstract how a neighbor vertex is accessed, given an object obtained from traversing the neighbor list.

- If variable `G` is of type `Graph` and variable `e` is of the value type of the inner range of `Graph`, then `target(G, e)` is a valid expression that returns a type of `vertex_id_t<Graph>`.
- All values returned by `target(G, e)` must be able to correctly index into a `Graph G`.

With this abstraction, the loop and neighbor access in `bfs` and `dijkstra` (respectively at lines 8 and 9 of Fig 2) are replaced by

```
for (auto&& e : G[u]) {
    auto v = target(G, e);
    ... }

```

Now, provided that suitable overloads for `target` are defined, the two model graph types

```
std::vector<std::vector<int>>;
std::vector<std::vector<std::tuple<int, float, double>>>;

```

will satisfy the above requirements, and we can compose them with the `bfs` and `dijkstra` we have lifted to this point. We can, for example, define overloads for `target` thusly:

```
int target(const std::vector<std::vector<int>>& G, int e) { return e; }
using E = std::tuple<int, float, double>;
int target(const std::vector<std::vector<E>>& G, E& e) { return std::get<0>(e); }

```

Note that these overloads are each specific to a single graph type. In practice we can define generalized overloads for entire classes of containers. In `NWGraph` we opted to realize `target` as a CPO, implemented using the `tag_invoke` mechanism [3].

4.3.3 Encapsulating Lifted Requirements as Concepts

We can encapsulate (and formalize) the above requirements in the form of a concept (which is almost a direct translation of the stated requirements to C++ code).

We first capture the very fundamental requirements of a graph, that it is a `semiregular` type (meaning that it is copyable and default-constructible) and that it has an associated `vertex_id_t` type:

```
template <typename G>
concept graph = std::semiregular<G>
    && requires(G g) { typename vertex_id_t<G>; };

```

We define this as a separate concept since we may wish to define other concepts that reuse these requirements.

Next, we define some convenience type aliases to capture the type of the inner range of a graph as well as the type that is stored by the inner range:

```
template <typename G>
using inner_range = std::ranges::range_value_t<G>;
template <typename G>
using inner_value = std::ranges::range_value_t<inner_range<G>>;

```

Now we can define the concept that captures the requirements from the lifted `bfs` and lifted `dijkstra`:

```
template <typename G>
concept adjacency_list = graph<G>
    && std::ranges::random_access_range<G>
    && std::ranges::forward_range<inner_range_t<G>>
    && std::convertible_to<vertex_id_t<G>, std::ranges::range_difference_t<G>>
    && requires(G g, vertex_id_t<G> u, inner_value_t<G> e) {
{ g[u] } -> std::convertible_to<inner_range_t<G>>;
{ target(g, e) } -> std::convertible_to<vertex_id_t<G>>; };

```

31:12 NWGraph

Although we restricted our illustration of lifting to `bfs` and `dijkstra` in this paper, this concept captures the requirements for all algorithms in `NWGraph` based on adjacency lists (see also the discussion in Section 4.4).

We can use this concept to constrain the interface to `bfs` in the following two ways:

```
template <class Graph>
requires adjacency_list<Graph>
void bfs(const Graph& G);
```

```
template <adjacency_list Graph>
void bfs(const Graph& G);
```

When using concepts via the `requires` keyword, there is usually a fully general declaration and a number of abbreviated forms. In `NWGraph`, the second syntax above is preferred.

4.3.4 Lifting Edge Weight

In the concrete implementation of Dijkstra's algorithm shown above, we assumed the container associated with each vertex in the graph (i.e., the container obtained by `G[u]`) provided tuples containing the vertex id and the edge weight. In fact, there are numerous ways to associate a weight with each edge. We could, for example, store an edge index with each neighbor and use that to index into an array that we also pass into `dijkstra`. In such a case the (unconstrained) prototype for the algorithm might be

```
template <class Graph, class Range>
auto dijkstra(const Graph& G, vertex_id_t<Graph> s, Range wt);
```

The inner loop might then look like

```
for (auto&& e : G[u]) {
    auto v = target(e);
    auto w = wt[v];
    if (d[u] + w < d[v]) {
        d[v] = d[u] + w;
        pi[v] = u;    }}
```

To lift this version and the version with the directly-stored property on edges, we introduce `weights` as a parameter at the interface of `dijkstra`, of parameterized type `WeightFunction`. The `WeightFunction` template parameter is constrained by the `std::invocable` concept, which specifies that the function must be callable on an argument of the `inner_value` type of the `Graph`.

```
template <adjacency_list Graph, std::invocable<inner_value<Graph> WeightFunction>
auto dijkstra(const Graph& G, vertex_id_t<Graph> s, WeightFunction wt);
```

In this case, the inner loop would look like

```
for (auto&& e : G[u]) {
    auto v = target(e);
    auto w = wt(e);
    if (d[u] + w < d[v]) {
        d[v] = d[u] + w;
        pi[v] = u;    }}
```

4.3.5 About Vertex IDs

There is one aspect of the NWGraph `adjacency_list` concept that may seem overly restrictive, namely that the outer range of an `adjacency_list` be a random-access range and, hence, indexable by values in the range $[0, |V|)$. There are two reasons for this particular design decision. First, indexing into the outer range (`g[u]`) must be a constant-time operation in order for algorithms using `g[u]` to have their expected computational complexity (which is part of an algorithm's specification). Second, vertex ids are used not just for indexing into the graph itself, but for accessing vertex properties, which we also expect to be random-access ranges. This does not, however, necessarily imply that graph inner ranges must store vertex ids. Rather, the `adjacency_list` concept only requires that the `target` CPO return something convertible to a `vertex_id_t`, something that can be computed or looked up (though, again, in constant time). However, if one is going to compute a `vertex_id_t` on the fly, or look it up elsewhere, one could as well store it. NWGraph containers take this approach, and it can also be readily realized by nested standard library containers (e.g., `std::vector<std::vector<int>>`).

That all being said, the NWGraph `adjacency_list` constraints (like all concepts) are only syntactically enforced. Though unnecessary, as described above, one could provide a graph that used an `std::map` as the outer container. The operation `g[u]` would still work, but at the cost of increased computational complexity.

4.3.6 Non-Type Constraints

We have already seen in lifting the edge weight that not all constraints for an algorithm are encapsulated in the type requirements for the input graph. There are other requirements that an algorithm may have that cannot be captured as a type requirement, or as any compile-time checkable requirement. For example, some algorithms, such as triangle counting, may require that the edges within each neighborhood be sorted. Such requirements become part of the specification of the API, but cannot be made part of type checking. This is similar to, say, `binary_search` in the C++ standard library, which requires that the elements of the container to which it is applied be sorted. Yet, there is no such thing as a sorted container type in the standard library.

4.4 Other Graph Concepts

Our presentation thus far has developed a single concept (`adjacency_list`) and the reader may ask how broad that concept is, given the wide variety of potential graph algorithms. In fact, the `adjacency_list` concept is surprisingly broad in its applicability; only a few supplemental concepts are required to cover all of the algorithms implemented in NWGraph and probably all of the algorithms that are likely to be implemented in NWGraph in the future. This is perhaps not so surprising since the adjacency list $Adj(G)$ is also the primary theoretical construct upon which the majority of graph algorithms are built.

There are two additional concepts that we introduce briefly here which we found necessary for algorithms in NWGraph: `degree_enumerable` and `edge_list`. The former extends `adjacency_list` with the requirement that there be a valid expression `degree`, necessary in some algorithms. The latter is basically a container of objects for which `source` and `target` are valid expressions. Algorithms such as Bellman-Ford and Kruskal's MST use an edge list rather than an adjacency list [8].

Our confidence that these few concepts are sufficient is based on a comprehensive study of the concepts in the Boost Graph Library (BGL) [29]. The BGL has five essential graph concepts that cover all of its algorithms: `VertexListGraph`, `EdgeListGraph`, `AdjacencyGraph`, `IncidenceGraph`, and `BiIncidenceGraph`. Of these, the design decisions of NWGraph to require vertex identifiers to be indices obviates `VertexListGraph`; we don't need to iterate through a list of vertices provided by the graph, we simply iterate through vertex ids from 0 through $|V| - 1$. The NWGraph `adjacency_list` and `degree_enumerable` concepts subsume the essential functionality of `AdjacencyGraph` and `IncidenceGraph`. The `adjacency_list` does not have a `source` function requirement, but that is in fact only rarely used in the BGL algorithms requiring `IncidenceGraph` (and when it is used, there are other ways of obtaining the same information). The `BiIncidenceGraph` concept specifies that a graph type must have two lists of neighbors: those reachable by “out edges” (which is what `adjacency_list` requires) and those that can reach the vertex, i.e., the “in edges.” The in edge neighborhoods are essentially the transpose of the out edge neighborhoods and can be represented with the same kind of adjacency list structure as the out edge neighborhoods. The need for a single data type holding lists of both out edges and in edges is unnecessary in the NWGraph design. Algorithms requiring a graph and its transpose take two graph arguments, one representing the out edges and one representing the in edges. Those two graphs represent (and store) exactly the same information as would be contained in a single `BiIncidenceGraph`, so there is no loss of efficiency in this design decision (and, in fact, NWGraph provides utilities for creating the transpose of a given graph). Finally, NWGraph includes an `edge_list` concept which is identical to the BGL `EdgeListGraph` concept.

5 Algorithms in NWGraph

Algorithms in NWGraph constitute the core of our library. NWGraph includes a broad classes of algorithms (sequential and parallel) for different graph problems, including graph traversal (BFS, SSSP), analytics (PageRank, Jaccard similarity, betweenness centrality, connected components), motif counting (triangle counting), network flow (maximum flow), etc. Table 1 lists the graph algorithms implemented in NWGraph along with their problem definitions.

5.1 Parallelization

NWGraph leverages existing parallelization support in the C++ standard library for implementing different parallel graph algorithms. However, in cases where it was necessary to circumvent some of the limitations of the C++ standard library for parallelization, we instead used Intel[®] oneAPI Threading Building Blocks (TBB) [14] for better performance.

5.1.1 Parallelization with `std` Execution Policies

NWGraph implements parallel algorithms for some of the different graph kernels described in Table 1 with `std::execution::par` (parallel policy) and `std::execution::par_unseq` (parallel unsequenced policy) provided to the `std::for_each` construct. Listing 5 demonstrates a triangle counting algorithm capable of benefiting from parallel `std::execution` policies. Note that updating shared variables relies on the `std::atomic` operations library.

Alternatively, Listing 6 shows an asynchronous task-based parallel triangle counting algorithm, which uses `std::future` and `std::async` to explicitly manage concurrency.

■ **Table 1** Algorithm classes in NWGraph. Parallel implementation available: `std::execution` and `std::for_each†`, `std::async§`, TBB's `parallel_for¶`.

Algorithm	Definition
Breadth-first search ^{†¶}	Traverses a graph in breadth-first search order from a given source. Implementation includes: top-down, bottom-up and direction-optimized [5] algorithms.
Depth-first search	Traverses a graph in depth-first search order from a given source.
Single-source shortest paths ^{†¶}	Finds the shortest distance paths from a given source to all other vertices in a graph. Δ -stepping algorithm [19] is implemented.
Connected component ^{†¶}	Finds connected components in a graph. Implementations include Afforest [32], Shiloach-Vishkin [26], BFS-based [27] and minimal label propagation [24, 34] algorithms.
PageRank ^{†§¶}	Compute the importance of each vertex in a graph. Implements the Gauss-Seidel algorithm [1].
Triangle counting ^{†§¶}	Counts the number of triangles in a graph. Implements algorithms discussed in [18].
Betweenness centrality ^{†§¶}	Measures how many times each vertex lies on the shortest paths to other vertices. Brandes Algorithm [7] has been implemented.
Maximum flow	Given a source and a sink, find paths with available capacity and push flow through them until there are no more paths available. Implements Edmonds-Karp algorithm.
K-core	Finds the subgraph induced by removing all vertices with degree less than k.
Jaccard similarity	Computes the Jaccard similarity coefficient of each pair of vertices in a graph.
Graph coloring	Assign a color to each vertex in the graph so that no two neighboring vertices have the same color. Implements Jones-Plassmann algorithm [15].
Maximal independent set	Graph coloring with two colors.

■ **Listing 5** Parallel triangle counting algorithm with `std::execution` policies.

```

1  template <adjacency_list_graph Graph, class OuterExecutionPolicy =
2      std::execution::parallel_unsequenced_policy,
3      class InnerExecutionPolicy = std::execution::sequenced_policy>
4  std::size_t triangle_count(const Graph& A, OuterExecutionPolicy&& outer = {},
5      InnerExecutionPolicy inner = {}) {
6      std::atomic<std::size_t> total_triangles = 0;
7      std::for_each(outer, A.begin(), A.end(), [&](auto&& x) {
8          std::size_t triangles = 0;
9          for (auto &&i = x.begin(), e = x.end(); i != e; ++i) {
10             triangles += nw::graph::intersection_size(i,e,A[std::get<0>(*i)], inner);
11         }
12         total_triangles += triangles;
13     });
14     return total_triangles;
15 }

```

■ **Listing 6** Parallel triangle counting algorithm with `std::async`.

```

1  template <class Op>
2  std::size_t triangle_count_async(std::size_t threads, Op&& op) {
3      std::vector<std::future<size_t>> futures(threads);
4      for (std::size_t tid = 0; tid < threads; ++tid) {
5          futures[tid] = std::async(std::launch::async, op, tid);
6      }
7      // Reduce the outcome ...
8  }
9  template <adjacency_list_graph Graph>
10 std::size_t triangle_count_v2(const Graph& G, std::size_t threads = 1) {
11     auto first = G.begin();
12     auto last  = G.end();
13     return triangle_count_async(threads, [&](std::size_t tid) {
14         std::size_t triangles = 0;
15         for (auto i = first + tid; i < last; i += threads) {
16             for (auto j = (*i).begin(), end = (*i).end(); j != end; ++j) {
17                 // ...
18             }
19         }
20     });

```

5.1.2 Shortcomings of `std` Execution Policy-based Parallelization

The current `std::execution` and `std::thread` libraries lack adequate support for implementing efficient parallel graph algorithms. Some of the most important limitations include:

- Programmers do not have control over workload distribution or partitioning of data or work among threads.
- Thread-safe data structures are not part of the standard library. Having to manually use coarse-grained locking `lock` and `mutex` to make standard library containers thread-safe is labor-intensive and may severely limit the performance of parallel graph algorithms.
- Granularity of concurrency cannot be directly managed.

5.1.3 Parallelization with Intel® Threading Building Blocks

To circumvent these shortcomings, NWGraph leverages Intel® Threading Building Blocks (TBB) library. TBB provides a set of efficient concurrent containers (`hashmap`, `vector`, and `queue`) implemented with fine-grained locking and lock-free techniques. NWGraph uses TBB's concurrent vector to maintain the frontier list of active vertices in each step of the Δ -stepping algorithm [19] for computing SSSP (Listing 7).

One determinant of parallel graph algorithm performance is how well the parallel workload is balanced among threads. Graph algorithms typically do not perform well with naive partitioning approaches. Recall a graph structure is a random-access range of forward ranges. A naive partitioning scheme will partition the outer range into equal-sized chunks – which is a reasonable strategy for one-dimensional containers, where each partition will have essentially the same amount of work. The story is completely different for graph data structures, especially those with highly skewed degree distributions, such as power-law graphs. In such cases, if the graph is partitioned based on the outer range, each partition will have the same number of starting vertices (the same number of inner ranges), but the number of neighbors

■ **Listing 7** Δ -stepping algorithm for computing single-source shortest paths using TBB’s thread-safe containers.

```

1  template <class distance_t, adjacency_list_graph Graph, class Id, class T>
2  auto delta_stepping(const Graph& graph, Id source, T delta) {
3      tbb::queuing_mutex                lock;
4      tbb::concurrent_vector<tbb::concurrent_vector<Id>> bins(size);
5      tbb::concurrent_vector<Id> frontier;
6      // ...
7      while (top_bin < bins.size()) {
8          frontier.resize(0);
9          std::swap(frontier, bins[top_bin]);
10         tbb::parallel_for_each(frontier, [&](auto&& u) {
11             if (tdist[u] >= delta * top_bin) {
12                 nw::graph::parallel_for(graph[u], [&](auto&& v, auto&& wt) {
13                     relax(u, v, wt); });
14             } });
15         // ...
16     }
17 }

```

■ **Listing 8** Δ -stepping algorithm for computing single-source shortest paths using TBB’s `blocked_range` partitioning technique.

```

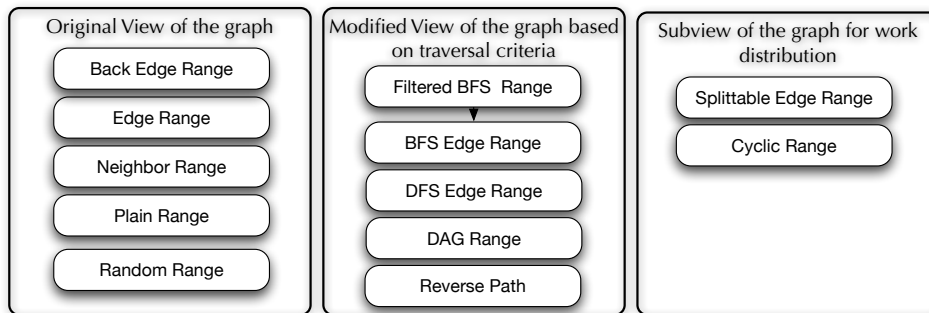
1  // ...
2  while (top_bin < bins.size()) {
3      // ...
4      tbb::parallel_for(tbb::blocked_range(0ul, frontier.size()), [&](auto&& range){
5          for (auto id = range.begin(), e = range.end(); id < e; ++id) {
6              auto i = frontier[id];
7              if (tdist[i] >= delta * top_bin) {
8                  // ...
9              }
10         }
11     });
12 }

```

in each inner range will vary with the degree distribution. Without the ability to partition based on the size of the inner ranges (which is an indication of the amount of work to be done for each partition), some threads may end up with vastly more work than other threads.

To provide better control of workload distribution among threads, TBB’s `parallel_for` function accepts ranges (a TBB construct in this case, not to be confused with C++20 ranges) that can be customized to provide user-defined partitioning. An example is NWGraph’s use of TBB’s `blocked_range` in the Δ -stepping algorithm (compare Listing 7 with Listing 8). Custom ranges are not limited to contiguous partitions of the underlying data structure. Instead, one can use strided (or cyclic) partitions – or, more generally, block-cyclic partitions – which can provide natural load balancing in certain situations. We show the performance benefit of cyclic distribution in Section 8.

TBB also implements C++ standard library parallelism (TBB’s parallel STL). Intel has open-sourced TBB (now called oneTBB), allowing its parallel STL effecting parallelism in other C++ library implementations. In fact, and somewhat ironically, the standard library provided by g++ (the compiler we used for NWGraph development) is one such compiler that uses TBB under the hood.



■ **Figure 3** Range adaptors in NWGraph.

6 Graph Range Adaptors in NWGraph

A key feature of the new C++ Ranges is the notion of *views*, which allow for different ways to access data in a range without changing the underlying range. Between a range and a range view sits a *range adaptor*, which takes the original range and presents it to the user as a view while hiding the underlying data manipulation details. We leverage range adaptors to simplify graph algorithms in NWGraph, by providing reusable data access patterns that eliminate the need for visitor objects.

Consider again BFS traversal, a core graph algorithm kernel. Except perhaps for benchmarking, a standalone BFS traversal is rarely useful. Rather, other algorithms use a BFS traversal pattern to perform more useful computations, such as finding the distance to every vertex from the source, finding the parent list, etc. One approach to applying BFS traversals to other types of computations would be to further parameterize `bfs` with additional functions. However, what to apply and where to apply it is not well defined – we don’t necessarily have well-defined concrete algorithms to lift.

The Boost Graph Library provides extensibility to BFS through its *Visitor* mechanism, which is essentially a large structure with callbacks used at multiple entry points in BFS execution [29]. The BFS Visitor has nine different possible callbacks, making actual extension of the BGL BFS a complicated proposition.

NWGraph does not attempt to further lift algorithms from arbitrary, concrete use cases (which are not well-defined from a library designer’s perspective). Instead it provides range adaptors that allow the graph to be iterated over in a specified order (either vertex by vertex or edge by edge). For example, NWGraph provides `bfs_range` for traversing the vertices of a graph in breadth-first order, and `bfs_edge_range` for traversing the edges.

```
for (auto&& u : bfs_range(G)) { /* visit vertex u */ }
for (auto&& [u, v] : bfs_edge_range(G)) { /* Visit edge u,v */ }
```

As views are concise and efficient ways of representing the same data in multiple ways, graph algorithms can be considered as operating on a range of elements of a graph with different requirements on how data is being viewed by the algorithm. In NWGraph, we provide three categories of view of the graph shown in Figure 3:

- *Original view of the graph:* These include edge range, neighbor range, plain range, random range and back-edge range.
- *Modified view of the graph based on traversal criteria:* For example, BFS and DFS traversal-based algorithms consider vertices in a certain order. These alternative views include BFS edge range, filtered BFS range, DFS edge range, Directed Acyclic Graph

(DAG) range and Reverse Path. More sophisticated range adaptors such as DAG range, for example, iterate over the vertices in a particular order, based on the predecessor-successor relationships, imposed by algorithm-specific heuristics.

- *Subview of the graph for workload distribution in parallel execution:* These include splittable edge range and cyclic range.

7 Model Data Structures in NWGraph

In Section 4.3 (lifting), we demonstrated that the built-in types in the standard library are sufficient to construct a graph. We reiterate that any data structure meeting the requirements specified by the NWGraph concepts can be composed with the NWGraph algorithms based on those concepts. For instance, `std::vector<std::vector<std::tuple<size_t, double>>>` meets the requirements of the `adjacency_list` concept, and hence can be used with any of the appropriate algorithms. Graphs do not need to be constructed *from* a range of ranges in order to meet the requirements *of* a range of ranges. Data structures such as compressed sparse structures, which represent all of a graph's neighborhoods contiguously in memory, can offer better performance due to more favorable memory accesses. We compare compressed sparse structures to compositions of standard library components in Section 8.

The workhorse graph structure for NWGraph is the class template `nwgraph::adjacency`, a compressed structure with the following (abbreviated) interface:

```
template <int idx, class Attributes...>
class adjacency {
    class outer_iterator {
        using iterator_category =
            std::random_access_iterator_tag; ..};
    class inner_iterator;
    outer_iterator begin();
    outer_iterator end();
    operator[](index_t i) const; };
```

`nwgraph::adjacency` is parameterized on the types of the edge properties, using variadic template parameter `Attributes`, to allow an arbitrary number of edge properties of arbitrary type. The `idx` parameter is a hint indicating whether the adjacency structure is representing the out edges or the in edges of the edge list from which it was built. To allow `nwgraph::adjacency` to meet the requirements of `adjacency_list` (range of ranges), we define a private iterator type that acts as a random access iterator.

NWGraph has a small set of utility functions for building graphs from a given dataset in a generic fashion. The first step involves building an index edge list, given a vertex table and an edge table. The second step uses this edge list to build an index graph (that is, filling in a structure modeling adjacency). Some algorithms (such as triangle counting) require sorted data in the neighborhood range. The graph construction algorithms take a runtime flag that indicates whether neighborhood sorting should be done during graph construction. NWGraph also provides functions to sort graphs that have already been constructed. The pertinent APIs for graph construction are the following:

```
template <class IndexEdgeList, class VRange, class ERange>
IndexEdgeList make_index_edge_list(const VRange& vertices, const ERange& edges);
template <adjacency_list Graph, class IndexEdgeList>
Graph make_graph(const IndexEdgeList& edge_list);
```

8 Performance Evaluation

8.1 Experimental Setup

Our experiments were carried out on compute nodes consisting of two Intel® Xeon® Gold 6230 processors, each with 20 physical cores running at 2.1 GHz (with turbo boost up to 3.9GHz), and hyperthreading disabled. Each processor has 28MB L3 cache and 188GB of main memory. NWGraph is implemented in C++20, parallelized with oneTBB 2021.4, and compiled with the g++ 11.2 compiler using `-Ofast -march=native` compilation flags.

8.2 Abstraction Penalty

Modern C++ practice includes a wide variety of mechanisms and related idioms for traversing data structures. Since the inner range of a type meeting `adjacency_list` requirements is a forward range, any of those modern techniques may be used for traversal. Moreover, the compressed graph structure provided in NWGraph presents a facade of being a range of ranges, using internally-provided iterators to effect the “range of ranges” interface. Given this variety of traversal mechanisms, and the layers of abstraction associated with traversal and with the compressed graph structure, there is potential for unintended abstraction penalty.

To verify the performance expectation of specialization in generic libraries, i.e., that there is minimal abstraction penalty, NWGraph includes an abstraction penalty benchmark suite, from which we present a small subset. Here, we focus on inner range traversal as it is ubiquitous to all graph algorithms; any penalties uncovered there would also be apparent in other graph algorithms. We use the sparse matrix-vector product (SpMV) algorithm as the vehicle for our study, as it is well-suited for characterizing inner range traversal; it makes one pass through the entire graph, traversing each of the inner ranges.

Let us consider a “raw for loop” implementation of SpMV, using a compressed sparse row (CSR) data structure to store the adjacency list. The CSR structure stores its neighbor indices and edge weights in contiguous arrays and traverses the data structure by looping through each vertex id and then traversing the associated inner range delimited by the indices in the `ptr` array.

```

auto ptr = G.indices_.data();
auto idx = std::get<0>(G.to_be_indexed_).data();
auto dat = std::get<1>(G.to_be_indexed_).data();
for (vertex_id_t i = 0; i < N; ++i) {
    for (auto j = ptr[i]; j < ptr[i + 1]; ++j) {
        y[i] += x[idx[j]] * dat[j]; }
}

```

This concrete algorithm establishes the baseline performance against which the generic algorithms are compared.

In a generic SpMV implementation, we cannot assume this underlying CSR structure. Rather we can only assume the interface specified by the `adjacency_list` concept, i.e., a range of ranges, and our implementations of a generic SpMV must be written accordingly. However, to meet our specialization performance requirements, a generic SpMV written to the `adjacency_list` concept must still provide the same performance as the concrete baseline when composed with a CSR-like structure, i.e., the NWGraph compressed graph `adjacency` structure.

Consider two common iteration patterns used in modern C++, an iterator-based for loop and a range-based for loop (which is essentially syntactic sugar for the iterator-based loop):

```

vertex_id_t k = 0;
for (auto i = G.begin(); i != G.end(); ++i) {
  for (auto j = (*i).begin(); j != (*i).end(); ++j) {
    y[k] += x[get<0>(*j)] * get<1>(*j); }
  ++k; }

```

```

vertex_id_t k = 0;
for (auto&& i : G) {
  for (auto&& [j, v] : i) {
    y[k] += x[j] * v; }
  ++k; }

```

As generic loops, these can be applied to any graph that models the `adjacency_list` concept. There are several important departures from the concrete CSR-based loops. It is easy to see how these operate on something that is a range of ranges. On the other hand, there is no obvious correspondence between the iterator-based algorithms and the concrete algorithm. Of particular note is that the neighbor vertex index `j` and the edge weight `v` are accessed as tuples, directly in the former case and via structured binding in the second case.

Iterators can also be used to traverse the inner range using the standard library `std::for_each` algorithm rather than `for` loops. The `std::for_each` algorithm iterates through the indicated iterator range and applies a given function to each element in the range. Here, we specify those functions using C++ lambdas.

```

vertex_id_type k = 0;
std::for_each(graph.begin(), graph.end(), [&](auto&& nbhd) {
  std::for_each(nbhd.begin(), nbhd.end(), [&](auto&& elt) {
    auto&& [j, v] = elt;
    y[k] += x[j] * v; });
  ++k; });

```

In the previous examples, we iterate through the graph using two nested loops, variously expressed. We can alternatively use the `edge_range` range adaptor, which “flattens” the graph, allowing traversal of all of the inner ranges with a single loop.

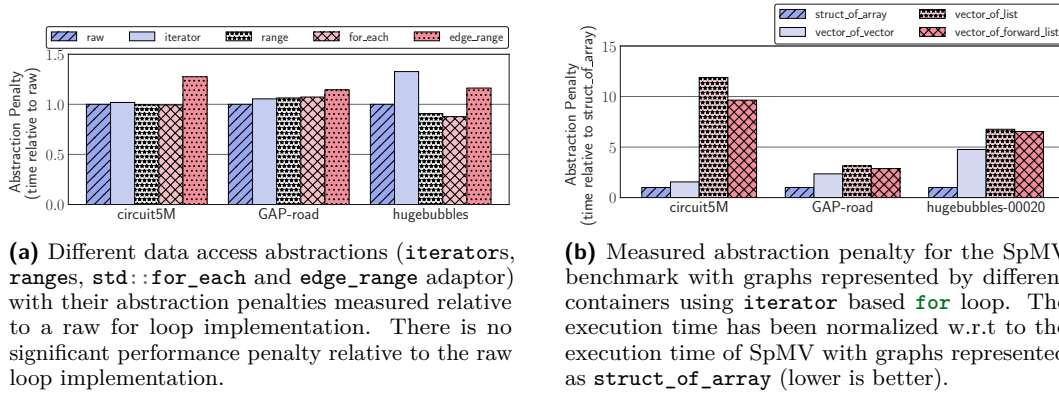
```

for (auto [i, j, v] : make_edge_range<0>(graph))
  y[i] += x[j] * v;

```

The `edge_range` adaptor essentially turns the `adjacency_list` into an `edge_list`. It provides a tuple with three elements: The source vertex, the target vertex, and the edge weight. The result is an extremely concise implementation of SpMV, which, again, will work with any type meeting the requirements of `adjacency_list`. The question that we wish to address is whether this genericity and this conciseness comes at the cost of performance.

Our experimental evaluation of SpMV uses three graphs with different underlying topology taken from the SuiteSparse matrix collection: `circuit5M`, `GAP-road`, and `hugebubbles` [9]. These graphs have similar numbers of edges (30M to 60M) and the benchmarks run in comparable time. Figure 4a shows the results of the different data access abstractions relative to the raw loop timing, for each benchmark. Timing results were averaged over 5 runs of each benchmark. Bars significantly higher than the raw for loop bar would indicate a significant performance penalty. None of the abstraction methods incurs a significant performance



■ **Figure 4** Abstraction Penalty Benchmarks with SpMV.

■ **Table 2** Characteristics of input graphs used for performance evaluation.

Name	Description	#Vertices (M)	#Edges (M)	Degree Dis- tribution	References
<code>road</code>	USA road network	23.9	57.7	bounded	[11]
<code>twitter</code>	Twitter follower Links	61.6	1,468.4	power	[17]
<code>web</code>	Web Crawl of .sk Domain	50.6	1,930.3	power	[6]
<code>kron</code>	Synthetic Graph	134.2	2,111.6	power	[20]
<code>urand</code>	Uniform Random Graph	134.2	2,147.5	normal	[12]

penalty relative to the raw loop implementation. `edge_range` is perhaps consistently a little higher than the baseline, due to moving access of the row index from the outer loop to the inner loop. Continued refinement of `edge_range` is a topic of ongoing work.

8.3 Graph Representations

We also evaluated the performance implications of different choices for the inner range: `adjacency`, `vector_of_vector`, `vector_of_list`, and `vector_of_forward_list`. The latter three graph structures are lightweight wrappers around the corresponding composed standard library containers, and provide a variadic interface to match `adjacency`. Note that all of these containers meet the requirement of our `graph` concept. However, they have different features outside of the context of graph algorithms that might make them suitable for different situations. Notably they can represent more dynamic graphs, i.e., they can be modified (vertices or edges added or deleted) much more efficiently than the compressed form.

This flexibility comes at a cost. Figure 4b shows the performance of the iterator-based SpMV on the different containers. Execution time is normalized relative to SpMV with the `adjacency` container. Unlike the results in Figure 4a, there are significant differences in performance between the different cases. Note, however, that these experiments are not measuring the difference between an abstract and a concrete expression of an algorithm. Rather, the generic algorithm is the same in each of the cases, but it is composed with different data structures. The benchmark compares the time it takes to traverse the different inner range structures (vector, doubly-linked list, singly linked list). The `adjacency` representation is cache-friendly, supporting efficient access of the outer and inner range, while the performance of the other graph types reflect the expected overheads of their underlying inner ranges.

■ **Table 3** GAP Benchmark Suite execution times for NWGraph.

Algorithm	kron	urand	twitter	web	road
BFS	0.51s	1.12s	0.26s	0.72s	0.84s
SSSP	7.23s	13.20s	2.88s	2.02s	2.99s
CC	0.64s	1.50s	0.29s	0.32s	0.09s
PR	12.09s	12.45s	8.69s	2.81s	0.26s
BC	8.43s	12.70s	2.42s	1.31s	1.95s
TC	305.19s	20.42s	58.81s	7.40s	0.07s

■ **Table 4** Performance comparisons with NWGraph for the GAP Benchmark Suite. Percentages represent the relative speedup of each particular experiment relative to the NWGraph. The color code indicates performance that is lower than (red), equal to (white), or higher than (green) NWGraph.

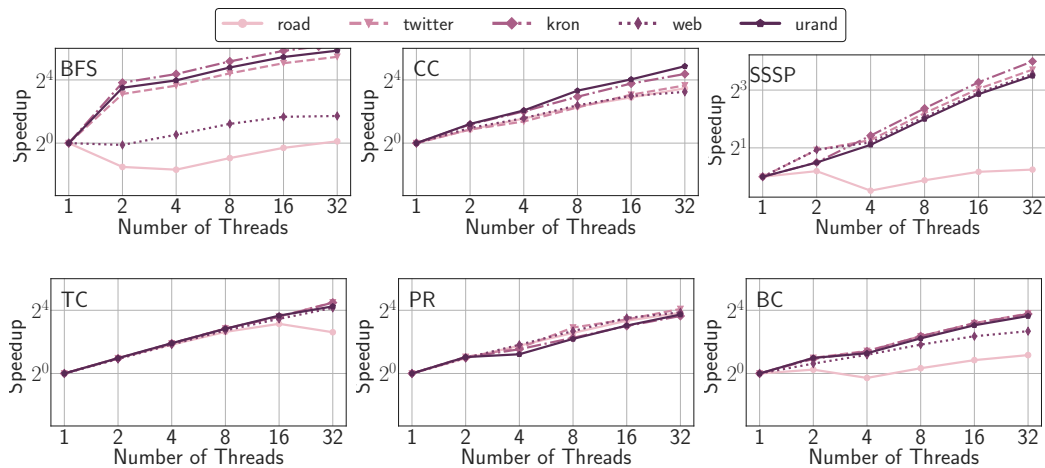
	Galois					GraphIt					GAPBS				
	kron	urand	twitter	web	road	kron	urand	twitter	web	road	kron	urand	twitter	web	road
BFS	118%	168%	40%	242%	886%	9.2%	9.5%	1.8%	8.2%	57%	174%	203%	168%	240%	218%
SSSP	136%	131%	108%	147%	1162%	100%	202%	114%	DNR	2.3%	189%	228%	250%	261%	1339%
CC	104%	90%	101%	174%	154%	0.34%	135%	0.53%	75%	0.02%	151%	112%	164%	174%	10%
PR	86%	65%	78%	99%	73%	4.5%	4.0%	4.6%	5.8%	4.8%	137%	116%	828%	101%	9.1%
BC	OOM	28%	0.08%	1.4%	44%	OOM	OOM	OOM	OOM	OOM	37%	38%	45%	40%	110%
TC	80%	71%	84%	56%	240%	107%	OOM	86%	OOM	120%	90%	98%	130%	54%	42%

8.4 Performance on Large-Scale Graphs

To demonstrate NWGraph’s performance characteristics on large-scale graphs, we evaluate and compare NWGraph with three well-established high-performance graph frameworks: GAP [4], Galois [22] and GraphIt [36], on the algorithms and graphs that comprise the GAP benchmark suite [4]. The algorithms in the benchmark are betweenness centrality (BC), breadth-first search (BFS), connected components (CC), PageRank (PR), single source shortest path (SSSP), and triangle counting (TC). The graphs used in the benchmark (shown in Table 2) are large, with diverse structural properties. All experiments were conducted with 32 threads running on 32 physical cores. The frameworks have been previously tuned for the GAP benchmark suite and were run under carefully controlled conditions, according to the rules and procedures established in [2].

Speedups of the different graph frameworks over NWGraph for the five datasets is shown in Table 4. We summarize our observations as follows:

- NWGraph outperforms the other frameworks in the majority of cases for BC and TC. The TC implementation has been highly tuned, using a cyclic range adaptor for effective load balancing, as well as having efficient implementations of its pre-processing techniques (which time is included in the benchmarking), such as relabeling the vertices by degree [18].
- NWGraph is better than Galois and GraphIt for PR, and somewhat worse than GAPBS. NWGraph and GAPBS both implement PR using a more efficient Gauss-Seidel inner step in the algorithm.
- For BFS and SSSP, NWGraph does not perform as well as Galois or GAPBS, particularly for *road*, for which Galois’s highly-asynchronous approach is particularly effective. We do not currently have an explanation for NWGraph’s poor performance on *road*.
- All frameworks except GraphIt implement the Afforest algorithm [32] for CC. Hence, GraphIt’s CC performs poorly for graph inputs having large dominant components.



■ **Figure 5** Strong scaling performance of six different graph algorithms (BFS, CC, TC, PR, PR, and BC) with five GAP graph inputs. The reported speedup is calculated as the ratio of the sequential (single-threaded) execution time and the parallel execution time.

One takeaway from these results is that the choice of algorithm and how well it is matched to a particular graph have the largest effect on performance. The performance differences between NWGraph and other frameworks (better or worse) are not due to inherent properties of the C++ language, nor its standard library, upon which NWGraph is built.

8.5 Strong Scaling Performance

Figure 5 presents the strong scaling performance for the graph kernels and inputs from the GAP benchmark [4]. For strong scaling, we keep the (size of the) dataset fixed while increasing the number of threads. The reported speedup is calculated by taking the ratio of the sequential execution time and the parallel execution time. In most cases, the algorithms scale well. Two exceptions can be observed with the road network input for the BFS and the SSSP algorithms. Since the road network has a low average degree and large diameter, increasing the number of threads does not improve the performance of these two algorithms significantly. BFS with web graph also does not demonstrate expected scalability.

8.6 Comparison with Boost Graph Library

We have compared NWGraph to BGL several times in this paper with respect to certain design decisions. Of interest also is how NWGraph performance would compare to BGL. We compare the sequential performance of the two libraries for four of the GAP graph kernels using the GAP graph inputs in Table 2. We report the results in Table 5. As can be observed from the Table, NWGraph performs better than BGL in all cases. (BGL has no directly comparable implementations of BC mor PR, and hence we are unable to compare the performance for these two kernels.)

9 Related Libraries and Toolkits

This section explores the landscape of related graph libraries and frameworks. Each of the libraries or tools discussed in this section make different design tradeoffs regarding usability, extensibility, and performance. Though few of the tools in this section (with the exception of BGL) aimed to fill the role of an STL graph library, they all contribute to a greater understanding of graph library design.

■ **Table 5** Sequential runtime and speedup of NWGraph and BGL for four graph algorithms: TC, CC, BFS, and SSSP. >24H indicates jobs that did not finish within 24 hours; OOM indicates out of memory.

Algorithm	Library	road	twitter	kron	web	urand
TC	BGL	1.34s	>24H	>24H	>24H	4425.54s
	NWGraph	0.41s	1327.63s	6840.38s	131.47s	387.53s
	Speedup	3.27	-	-	-	11.42
CC	BGL	1.36s	21.96s	81.18s	6.64	134.23
	NWGraph	1.02s	3.65s	13.37s	3.02s	43.74s
	Speedup	1.34	6.02	6.07	2.20	3.07
BFS	BGL	1.09s	12.11s	54.80s	5.52s	73.26s
	NWGraph	0.91s	11.25s	38.86s	2.37s	64.63s
	Speedup	1.20	1.08	1.41	2.33	1.13
SSSP	BGL	4.03s	47.89s	167.20s	28.29s	OOM
	NWGraph	3.35s	40.94s	95.06s	23.51s	177.13s
	Speedup	1.21	1.17	1.76	1.20	-

Generic C++ Graph Libraries. BGL [29] and the LEMON graph library [10] both contributed to the development of generic graph algorithms in C++. BGL proposed algorithm templates that could be used on a variety of graph types (which could be generated using BGL’s graph type generator), e.g., vector of lists, list of vectors, etc. Vertices and edges were allowed to be arbitrary types accessed via property maps which could be stored internally or externally to the graph. The default graph algorithms could be customized using visitor objects, which allowed users to use existing data access patterns to do additional work. LEMON shared many of these features. Both libraries advertise algorithms that work with user-defined graphs, so long as they conform to a certain interface.

Some of these features had shortcomings that limited their use. The visitor objects are difficult to use, both from a programming and algorithmic design perspective. Property maps are a powerful programming abstraction, but in addition to being difficult to use, could lead to performance issues. The type of LEMON’s graph adaptors are different from the original graph type being adapted, and their use as graphs is only supported in limited ways. A major shortcoming of these designs is the difficulty of using custom data structures. In order to adapt an existing user-defined data structure, the BGL interface requires overloading several global free functions. These mostly include accessors, mutators, and iterators for edges and vertices. An assumption is placed on the graph container type being adapted that it will have much of the same behavior as the built in BGL container types. Furthermore both libraries lack newer features in C++ such as `constexpr`, variadic templates, automatic type deduction, execution policies, etc.

HPC Graph Frameworks. There are several graph frameworks designed to maximize performance in distributed memory or shared memory, such graph frameworks include Parallel Boost Graph Library (PBGL) [13], Galois [16], Ligra [28], Giraph [25], Gunrock [33], GraphIt [35], etc. The contributions of these frameworks are typically a computational model for parallel processing of graphs, including on clusters and GPUs, with less emphasis on the usability or extensibility of graph algorithms or containers. A thorough evaluation of several well-known parallel graph frameworks can be found in [2].

10 Conclusion

In this paper we presented the design and rationale for a modern generic C++ library of graph algorithms and data structures, NWGraph. Based on a careful analysis of the graph problem domain, the fundamental interface abstraction underlying NWGraph is that of a random access range of forward ranges. Intentionally minimal, this interface admits composition with any types that meet its requirements. The library implementation includes selected concreted containers and a rich selection of common graph algorithms. Though the library is implemented with standard library components using idiomatic C++, experimental results showed that the interfaces present no abstraction penalty and that the NWGraph implementation has performance on par with the highest performing competition. We intend to continue to refine NWGraph and use it as a testbed in support of an emerging proposal to the C++ standards committee for a standard C++ graph library.

References

- 1 Arvind Arasu, Jasmine Novak, Andrew Tomkins, and John Tomlin. PageRank computation and the structure of the web: Experiments and algorithms. In *WWW*, pages 107–117, 2002.
- 2 Ariful Azad, Mohsen Mahmoudi Aznaveh, Scott Beamer, Mark Blanco, Jinhao Chen, Luke D’Alessandro, Roshan Dathathri, Tim Davis, Kevin Deweese, Jesun Firoz, et al. Evaluation of graph analytics frameworks using the gap benchmark suite. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*, pages 216–227. IEEE, 2020. doi:10.1109/IISWC50251.2020.00029.
- 3 Lewis Baker, Eric Niebler, and Kirk Shoop. tag_invoke: A general pattern for supporting customisable functions. Technical Report P1895R0, JTC1, 2019. URL: <http://open-std.org/JTC1/SC22/WG21/docs/papers/2019/p1895r0.pdf>.
- 4 Scott Beamer, Krste Asanović, and David Patterson. The GAP Benchmark Suite. *arXiv*, 2015. doi:10.48550/ARXIV.1508.03619.
- 5 Scott Beamer, Krste Asanović, and David A. Patterson. Direction-optimizing breadth-first search. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–10, 2012. doi:10.1109/SC.2012.50.
- 6 Paolo Boldi and Sebastiano Vigna. The WebGraph framework I: Compression techniques. *WWW*, pages 595–601, 2004. doi:10.1145/988672.988752.
- 7 Ulrik Brandes. A faster algorithm for betweenness centrality. *The Journal of Mathematical Sociology*, 25(2):163–177, 2001. doi:10.1080/0022250X.2001.9990249.
- 8 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. MIT Press, 3rd ed edition, 2009.
- 9 Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1), December 2011. doi:10.1145/2049662.2049663.
- 10 Balázs Dezső, Alpár Jüttner, and Péter Kovács. Lemon—an open source c++ graph template library. *Electronic Notes in Theoretical Computer Science*, 264(5):23–45, 2011. URL: <https://lemon.cs.elte.hu/trac/lemon>.
- 11 9th DIMACS implementation challenge - Shortest paths, 2006. URL: <http://www.dis.uniroma1.it/challenge9/>.
- 12 Paul Erdős and Alfréd Rényi. On random graphs. I. *Publicationes Mathematicae*, 6:290–297, 1959.
- 13 Douglas Gregor and Andrew Lumsdaine. Lifting sequential graph algorithms for distributed-memory parallel computation. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA ’05*, pages 423–437, New York, NY, USA, 2005. ACM. doi:10.1145/1094811.1094844.
- 14 Intel. Intel Threading Building Blocks (TBB), 2020. URL: <https://github.com/oneapi-src/oneTBB>.

- 15 Mark T Jones and Paul E Plassmann. A parallel graph coloring heuristic. *SIAM Journal on Scientific Computing*, 14(3):654–669, 1993. doi:10.1137/0914041.
- 16 Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *PLDI*, pages 211–222. ACM, 2007. doi:10.1145/1250734.1250759.
- 17 Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is twitter, a social network or a news media? In *Proceedings of the 19th International Conference on World Wide Web, WWW '10*, pages 591–600, New York, NY, USA, 2010. ACM. doi:10.1145/1772690.1772751.
- 18 Andrew Lumsdaine, Luke Dalessandro, Kevin Deweese, Jesun Firoz, and Scott McMillan. Triangle counting with cyclic distributions. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8, 2020. doi:10.1109/HPEC43674.2020.9286220.
- 19 Ulrich Meyer and Peter Sanders. Δ -stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1):114–152, 2003. 1998 European Symposium on Algorithms. doi:10.1016/S0196-6774(03)00076-2.
- 20 Richard C. Murphy, Kyle B. Wheeler, Brian W Barrett, and James A. Ang. Introducing the Graph 500. In *Cray User's Group*. CUG, 2010.
- 21 David R. Musser and Alexander A. Stepanov. Generic programming. In P Gianni, editor, *International Symposium ISSAC 1988*, volume 38 of *Lecture Notes in Computer Science*, pages 13–25. Springer-Verlag, 1989.
- 22 Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, SOSP '13*, pages 456–471, New York, NY, USA, 2013. ACM. doi:10.1145/2517349.2522739.
- 23 Eric Niebler, Casey Carter, and Christopher Di Bella. The one ranges proposal. Technical report, Tech. rep. P0896r4. Nov. 2018., 2018. URL: <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0896r4.pdf>.
- 24 S. M. Orzan. *On Distributed Verification and Verified Distribution*. Ph.d. thesis, VRIJE UNIVERSITEIT, November 2004. URL: <http://dare.ubvu.vu.nl/handle/1871/10338>.
- 25 Roman Shaposhnik, Claudio Martella, and Dionysios Logothetis. *Practical Graph Analytics with Apache Giraph*. Apress, New York, 1st ed. edition edition, October 2015.
- 26 Yossi Shiloach and Uzi Vishkin. An $O(\log n)$ parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57–67, 1982. doi:10.1016/0196-6774(82)90008-6.
- 27 J. Shun, L. Dhulipala, and G. Blelloch. A simple and practical linear-work parallel algorithm for connectivity. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '14*, pages 143–153. ACM, 2014. doi:10.1145/2612669.2612692.
- 28 Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '13*, pages 135–146, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2442516.2442530.
- 29 Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002.
- 30 Alexander Stepanov and Meng Lee. The standard template library. Technical Report HPL-95-11, HP Laboratories, November 1995. URL: <http://stepanovpapers.com/STL/DOC.PDF>.
- 31 Alexander Stepanov and Paul McJones. *Elements of Programming*. Addison-Wesley Professional, 1st edition, 2009.
- 32 Michael Sutton, Tal Ben-Nun, and Amnon Barak. Optimizing parallel graph connectivity computation via subgraph sampling. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 12–21. IEEE, 2018. doi:10.1109/IPDPS.2018.00012.
- 33 Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. Gunrock: A high-performance graph processing library on the gpu. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '16*, New York, NY, USA, 2016. ACM. doi:10.1145/2851141.2851145.

- 34 Da Yan, James Cheng, Kai Xing, Yi Lu, Wilfred Ng, and Yingyi Bu. Pregel algorithms for graph connectivity problems with performance guarantees. *Proc. VLDB Endow.*, 7(14):1821–1832, 2014. doi:10.14778/2733085.2733089.
- 35 Yunming Zhang, Ajay Brahmakshatriya, Xinyi Chen, Laxman Dhulipala, Shoaib Kamil, Saman Amarasinghe, and Julian Shun. Optimizing ordered graph algorithms with GraphIt. In *CGO*, pages 158–170. ACM, 2020. doi:10.1145/3368826.3377909.
- 36 Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. Graphit: A high-performance graph dsl. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018. doi:10.1145/3276491.