Pacific Northwest
NATIONAL LABORATORY

PNNL-32018

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

September 2021

Andrew Lumsdaine
Jesun S Firoz
Joseph B Manzano Franco
Andres Marquez
Joshua D Suetterlein
Marcin J Zalewski
Xu Tony Liu

U.S. DEPARTMENT OF ENERGY

Prepared for the U.S. Department of Energy
under Contract DE-AC05-76RL01830

DISCLAIMER

PACIFIC NORTHWEST NATIONAL LABORATORY
operated by
BATTELLE
for the
UNITED STATES DEPARTMENT OF ENERGY
under Contract DE-AC05-76RL01830

Printed in the United States of America

Available to DOE and DOE contractors from the Office of Scientific and Technical Information,
P.O. Box 62, Oak Ridge, TN 37831-0062
www.osti.gov
ph: (865) 576-8401
fax: (865) 576-5728
email: reports@osti.gov

Available to the public from the National Technical Information Service
5301 Shawnee Rd., Alexandria, VA 22312
ph: (800) 553-NTIS (6847)
or (703) 605-6000
email: info@ntis.gov
Online ordering: http://www.ntis.gov

PNNL-32018

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

September 2021

Andrew Lumsdaine
Jesun S Firoz
Joseph B Manzano Franco
Andres Marquez
Joshua D Suetterlein
Marcin J Zalewski
Xu Tony Liu

Prepared for
the U.S. Department of Energy
under Contract DE-AC05-76RL01830

Pacific Northwest National Laboratory
Richland, Washington 99354

Introduction

Graphs are powerful mathematical tools for reasoning about the relationships between given entities, focusing on the *characteristics and structures of the relationships*, independent of what the entities and the relationships actually are. Consequently, results from graph theory can be applied to any actual sets of data elements between which relationships can be established. This kind of generality – *genericity*, if you will – is a goal for software libraries as well as mathematical theories; we would like our algorithms to compose with any kind of data structure whose use with the algorithm makes sense. With the release of the celebrated Standard Template Library (Stepanov and Lee 1995), *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 relationship in a programmer's data.

Most graph libraries that one might find today are just that – libraries for graphs, for specific library-defined data structures, requiring programmers to convert their data into a specific graph datatype in order to use the library. To achieve the success of the STL, which supplanted the myriad container and utility algorithm libraries that came before it, a generic graph library will need the same level of composability. The Boost Graph Library (BGL) was an early attempt to develop a generic library of graph algorithms and data structures (Siek, Lee, and Lumsdaine 2002). An entire paper could be written just on a retrospective evaluation of the BGL, but the primary reason that we undertake here to present the development of yet another comprehensive generic library for graphs is that, despite its aspirations for genericity, the BGL did not quite escape being a library that was *for graphs* (and in particular, for BGL graphs), rather than being a library for arbitrary types representing entities and relationships.

In this paper we present 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*).

NWGraph contains the following innovations:

- A concept taxonomy (using C++20 concepts) for specifying graph algorithm requirements
- A graph is defined generically as a random access range of forward ranges
- A rich set of range adaptors

- An efficient and fully parallelized (using C++ execution policies) implementation
- An API designed to fully support modern idiomatic C++
- Maximum compatibility with third-party data structures and algorithms

In the following sections we first provide some basic background and terminology we will be using to discuss graph algorithms, as well as a bit more detail on generic programming. Next, we analyze the domain of graph algorithms with respect to common requirements and present the fundamental concepts in NWGraph. We then present an overview of the primary components of NWGraph in addition to its concepts: Its algorithms, containers, and adaptors. We present some examples of its use and composability and include the results taken from abstraction penalty experiments. Finally, we provide a high-level feature comparison of NWGraph with other extant graph libraries and conclude with some observations based on our experiences in developing NWGraph.

The complete source code and documentation for NWGraph are available on github at ⟨Redacted for double-blind review⟩.

1.0 Generic Programming, Concepts, and Ranges

Generic programming is a software development paradigm inspired by the organizational principles of mathematics. 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. The `iterator` concept taxonomy, for example, was the foundation upon which the STL was organized (Stepanov and Lee 1995; Musser and Stepanov 1989).

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 until as long as instantiation of the generic algorithm with concrete types remains as efficient as the equivalent concrete algorithm would have been. Generic libraries do not tolerate abstraction penalty.

It cannot be emphasized enough that in a generic library, the requirements on algorithms lead to the concepts, which in turn represent the interface to the library. The goal is to create an efficient framework of highly-reusable algorithms that can be composed with arbitrary third-party components – *not* to start with a data type intended to meet all needs (even in the guise of a concept) and then define conforming algorithms. Again, the library algorithms are primary.

1.1 C++20 Concepts

Interfaces to generic algorithms are expressed in terms of the properties of the types, instead of the types themselves. In this regard, concepts are the mechanism to define such properties or constraints on types. Concepts, one of the most salient features of C++20, define a family of allowable types for the generic components with valid expressions and associated types. A **`concept`** definition in C++20 declares a set of requirements on types. As long as these type requirements to an interface is met, developers and users can leverage the concept, while keeping the implementation details encapsulated. Imposing these requirements allows the compiler to type-check during compilation time and improves error detection. In other words, concepts are an extension of templates which help clarify which types can be used inside class and function template arguments. When used correctly, they also promise to a user that any type that meets these requirements can use the concept.

For example, the following `square` function requires that the `Scalar` type be an integer or a floating point type (**`concept`** `Number`). Attempting to call the function with a type that does not meet this concept requirement, such as a string, will yield a much more useful error message than a similar call that attempts to search for a missing multiplication operator.

```
template <typename T>
concept Number = std::integral<T> || std::floating_point<T>;
template <Number Scalar>
Scalar square(Scalar s) {
```

```
  return s*s;
}
square(5.2);
square("5.2"); // error: use of function 'Scalar square(Scalar)
//[with Scalar = const char*]' with unsatisfied constraints
```

Concepts also allows defining requirements on the interface and return types. For example, to restrict the `square` function to only compute the square value of an input greater than a certain number (`comp`), and for which the type `T` requires to define a `greater` function, concepts `BigNumber` and `isGreater` can be defined as follows (to replace the `Number` concept defined above).

```
template <typename T>
concept isGreater = requires (T t, T comp) {
  {t.greater(comp)} -> std::convertible_to<bool>;
};
template <typename T>
concept BigNumber = Number<T> && isGreater<T>;
```

Syntactically, here, the return type requirement is surrounded by braces ({}), followed by an arrow (->) and the constraint on the return type.

1.2 C++20 Ranges

The new C++20 Ranges library ((Niebler, Carter, and Di Bella 2018)) adds support for operating on ranges of elements. Simplistically ranges can be considered as an abstraction to a collection of items that can be iterated over. Ranges consists of a pair of begin and end iterators that are not required to be the same type. Ranges provide a way to make STL algorithms *composable* and improve the readability and writability of C++ code. An example of using `ranges` is:

```
std::vector<int> v { /* ... */ }

std::min_element(v.begin(), v.end());     // iterator interface
std::ranges::min_element(v);              // ranges interface
```

Range adaptors, alternatively known as *Views*, can be considered as wrappers around another range, without mutating or copying the original range.

Two range concepts are of particular interest. `ranges::forward_range` models iterating over a collection from the beginning to the end multiple times.
`ranges::random_access_range` allows indexing into a collection with `[]` operator in constant time.

2.0 Graphs

In order to describe the NWGraph library and its abstractions and interfaces, it is useful to examine in some more detail the particular problem domain of graphs and graph algorithms. We begin first with some terminology and then walk through (briefly) the abstraction process that we actually apply when thinking about problems in terms of graphs.

2.1 Graph Terminology

Abstractly, we define a graph G as comprising two finite sets, $G = \{V, E\}$, where the set V is a set of entities of interest ("vertices" or "nodes") and E is a set of pairs of entities from V ("edges" or "links"). Without loss of generality we label the entities in V as v_i so that $V = \{v_0, v_1, \ldots v_{n-1}\}$. The set of edges (also labeled) can be constructed using the labeled entities from V so that $E = \{e_0, e_1, \ldots e_{m-1}\}$. The edges may be ordered pairs, denoted as (v_i, v_j), which have equality defined such that $(v_i, v_j) = (v_m, v_n) \leftrightarrow v_i = v_m \wedge v_j = v_n$. Or, the edges may be unordered sets, denoted as $\{v_i, v_j\}$ which have equality defined as $(v_i, v_j) = (v_m, v_n) \leftrightarrow (v_i = v_m \wedge v_j = v_n) \vee (v_i = v_n \wedge v_j = v_m)$. If a graph is defined with ordered edges we say the graph is directed; if the graph is defined with unordered edges say the graph is undirected.

2.2 Graph Models

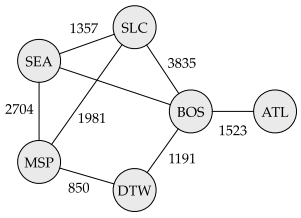
Graphs are powerful abstractions because they allow us to reason about the relationships between entities, irrespective of what the entities actually are. But, when we use graph algorithms in practice, we are using them to model some specific problem. Since one of the motivations behind NW Graph is to support graph computing in the context of real programs, we briefly describe the first part of the abstraction process when modeling with graphs.

Airport

SEA
MSP
SLC
DTW
ATL
BOS

Distance (km)

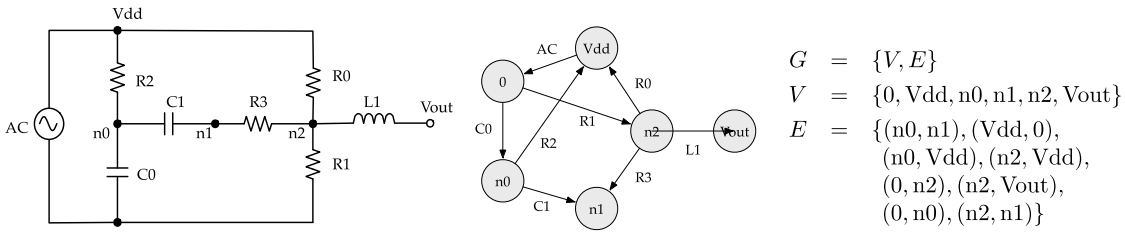
MSP	DTW	850
SLC	SEA	1357
MSP	SLC	1981
BOS	SLC	3835
SEA	BOS	4016
BOS	ATL	1523
SEA	MSP	2704
BOS	DTW	1191



$$
\begin{aligned}
G &= \{V, E\} \\
V &= \{\text{SEA}, \text{MSP}, \text{SLC}, \text{DTW}, \text{ATL}, \text{BOS}\} \\
E &= \{\{\text{MSP}, \text{DTW}\}, \{\text{SLC}, \text{SEA}\}, \\
&\quad \{\text{MSP}, \text{SLC}\}, \{\text{BOS}, \text{SLC}\}, \\
&\quad \{\text{SEA}, \text{BOS}\}, \{\text{BOS}, \text{ATL}\}, \\
&\quad \{\text{SEA}, \text{MSP}\}, \{\text{BOS}, \text{DTW}\}\}
\end{aligned}
$$

[fig:graph-model-airports] Airport route table modeled as an undirected graph.

Fig. 1 shows a model of an airport route table as an undirected graph. We begin with a table of airports and the distance in kilometers between pairs of them. We model this situation as a graph by identifying graph nodes with airports and graph edges with pairs of cities that are given as pairs in the distance table.



[fig:graph-model-circuit] Electrical circuit modeled as a directed graph.

Fig. 2 shows a model of an electrical circuit as a directed graph. Two-terminal circuit elements connect to each other at given nodes (also the terminology used in circuit modeling). We thus model circuit connection points as graph nodes, and the connections between them as edges. In the case of circuits, orientation of circuit elements matters and so we may choose (at least at this stage of the modeling process) to use directed edges in the graph.

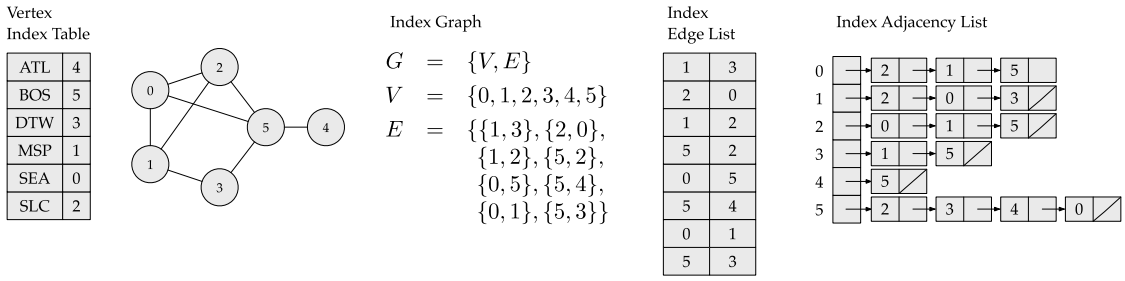
2.3 Representing Graphs

To define algorithms on graphs and to be able to reason about those algorithms, we need to define some representations for graphs—one can't really do very much with abstract sets of vertices and edges. So first we need to define some terminology regarding representations. Various 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 concepts.

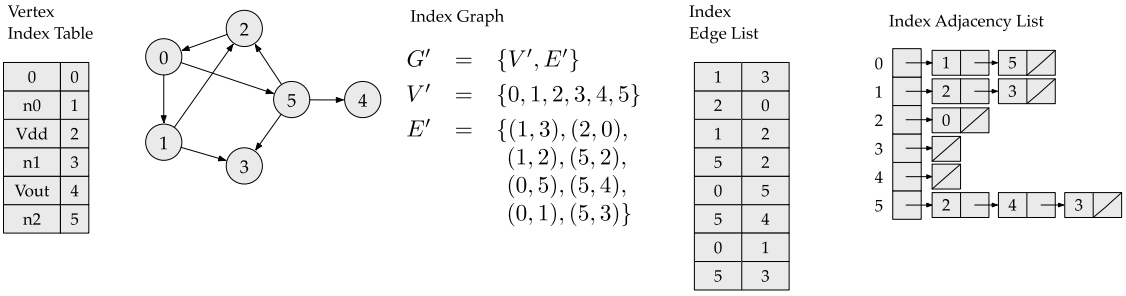
One of the fundamental operations in graph algorithms is *traversal*. That is, given a vertex u, we would like to find the *neighbors* of u, i.e., all vertices v such that the edge (u, v) is in the graph. Then, for each of those edges, we would like to find their neighbors, and so on. The representation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

There is an important transition in going from a graph (as a collection of vertex objects and pairs of vertex objects) to an adjacency list. Implied in using an adjacency list for traversal is that we would like to be able "find the neighbors" efficiently, i.e., in constant time, meaning we need to be able to take a vertex and do a constant time lookup to get a container of the neighboring vertices. Then, with what we get back as the neighbors, we also need to use to look up more neighbors. In short, regardless of what we consider to be the vertices or edges in our graph G, an adjacency list is something that stores indices which can be used to index into itself. Consequently, we need for the adjacency list to be something that is indexable.

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. Figs. 3 and 4 show the progression from an index graph to an index adjacency list (compare also to Figs. 1 and 2).



[fig:airport_index_to_adj] Index graph and associated index edge list and adjacency list corresponding to the airport graph example. Also shown is the translation table from vertex to index.



[fig:circuit_index_to_adj] Index graph and associated index edge list and adjacency list corresponding to the circuit graph example. Also shown is the translation table from vertex to index.

Of course, we don't need an underlying graph to define what an index graph itself is. We can say that a graph $G = (V, E)$ is an index graph if its vertex set is a set of contiguous indices, i.e., with $V = [0, |V| - 1)$. Since an index graph is just a graph, in cases where the context is clear, we may refer to an index graph simply as a graph. We note that an adjacency list can only be defined over an index graph.

Finally, we can make the following precise definition: An *adjacency list* of an index graph $G = (V, E)$ is an array Adj of size $|V|$ (the array is indexed from 0 to $|V| - 1$). Each entry $Adj[u]$ in the array is a container of all the vertices v for which (u, v) is contained 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. and show the index graph and the adjacency list representation of our airport and circuit examples.

Remark (1): Although the standard term for this kind of abstraction is "adjacency list", and although it is often drawn pictorially with linked lists as elements, it is not necessary that this abstraction be implemented as an actual linked list. 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): The index adjacency list does not store edges per se and therefore the index adjacency list is neither inherently directed nor undirected. That is, for a given vertex u, the container $Adj[u]$ contains the vertex v if the edge (u, v) is contained in E. This means that for a directed graph with edge (u, v) in , E, $Adj[u]$ will contain v. For an undirected graph with edge (u, v) is contained in E, $Adj[u]$ will contain v and $Adj[v]$ will contain u. Directedness of the

original graph is thus made manifest in the *values* stored in the index adjacency list. But there is nothing about the structure or semantic properties of the adjacency list itself that reflects the directedness (or undirectedness) of the original graph.

3.0 NWGraph: Core Library

3.1 Graphs as Ranges

Based on our observations of graphs and their representations, the foundational design decision in NWGraph is that the abstract interface presented by graphs is that of a *range of ranges*. The primary graph abstraction in the preceding discussion is an index adjacency list, which has the fundamental property that it stores things that can be used to index into itself. Accordingly, the index adjacency structure of a graph is modeled as *a random access range of forward ranges*. As such, the outer range conforms to the requirements of the `random_access_range` concept and the inner range conforms to the requirements of the `forward_range` concept, including all valid expressions and associated types (such as `begin`, `end`, etc.). The outer range is a range over the vertices, and the inner ranges are ranges over each vertex's neighbor edges. To access a vertex or edge property, a unique *key* or id is assigned to each vertex and edge, hereafter referred to as *Vertex key* and *Edge key* respectively. The outer range of a graph is indexed with a vertex key. Indexing into the outer range with a vertex key u returns an inner range, corresponding to the set of edges or vertices reachable from u, i.e., it contains information about all (u, v) in the edge set of the graph. The objects stored by the inner range associated with u can be accessed to obtain the source vertex key u, the target vertex key v, and (optionally) an edge key corresponding to the edge (u, v), or the properties associated with the edge (u, v).

Note that ranges in this case are not particular types but rather the description of the properties of types, i.e., a concept. Many concrete types can meet the conceptual requirements of a range of ranges, e.g., a `std::vector` of `std::vector`s. But it is important to realize that the requirements defined by concepts are on interfaces, types that are not containers of containers can still meet the concept requirements of a range of ranges. That is, a random access range is defined by the type of iterators it provides. In a range of ranges, those iterators would return a range when dereferenced, but that range is again determined by the type of iterators it provides, not by what type of storage it represents. These nested iterators can be realized with proxy classes rather than with nested containers.

As a result of the ranges of ranges abstraction, NWGraph is able to decouple graph algorithms from the underlying graph data structure implementations (such as tables, edge lists, adjacency matrices, etc.). The following code assumes the inner range stores pairs of vertices, corresponding to the edges leaving each vertex), and would iterate through all of the edges in the graph:

```
template <typename GraphT>
void view_edges(GraphT& G) {
  for (auto&& inner_range : G) {
    for (auto&& [u,v] : inner_range) {
      // do something
}}}
```

Note that a graph composed of standard library containers, e.g., `std:vector<std::forward_list<std::tuple<int, int>>>` or `std:vector<std::vector<std::tuple<int, int>>>` could be used in the example

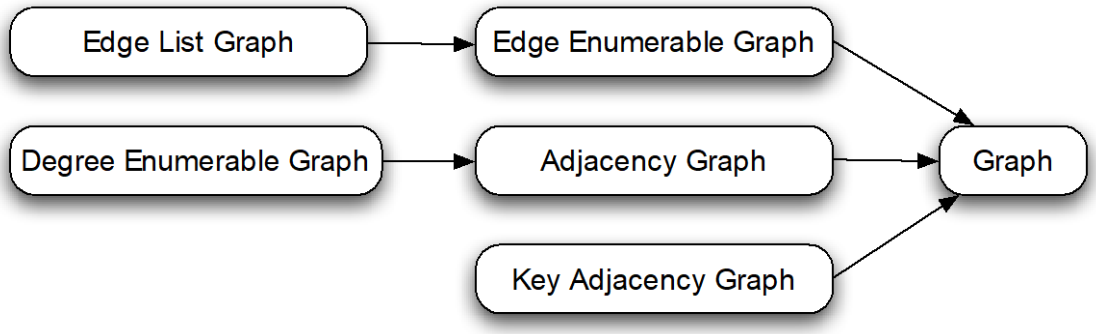
above. This is one of the most fundamental observations when considering graphs as range of ranges and consolidates our subsequent discussion about graph concepts, which essentially dictates that, as long as any container supplied to a graph algorithm is conformant to a desired range (of ranges) concept, it can be treated as a graph to be operated on.

3.2 Graph Concepts

Graph concepts are an essential tool in a composing third-party containers with NWGraph ("bring your own graph") because they specify an algorithm's requirements (no more, no less) of a custom user data-type. These are supported in NWGraph (as range of ranges in idiomatic C++) with control point objects and type functions. This allows third party containers to meet the concept requirements. Different graph algorithms will have very different requirements on the underlying ranges. Some graphs require modification via the adding or removal of vertices or edges. Many (but not all) algorithm currently implemented in NWGraph require random access over the outer range to have $O(1)$ access to a vertex's neighbor list. Such an algorithm would specify this requirement in the following way.

```
template <random_access_range GraphT>
void graph_algorithm(GraphT& G) {
    // Can access G.begin()[0]
}
```

We summarize the important concepts in NWGraph in . The relationship between the Graph concepts are depicted in .



Graph concept hierarchy in NWGraph.

[table:graph_concepts] Summary of graph concepts in NWGraph.

Concept	Description
`graph`	Specifies that a type is a graph, i.e., that it provides `vertex_id_type` and the number of vertices can be obtained in constant time
`edge_list_graph`	Specifies that a graph is a forward range of edges
`adjacency_graph`	Specifies that a graph is a random access range of forward ranges

Concept	Description
`key_adjacency_graph`	Specifies that a graph is a random access of pairs of vertex ids and forward ranges
`edge_enumerable_graph`	Specifies that the edge count of the graph can be returned in constant time
`degree_enumerable_graph`	Specifies that the degree of a vertex can be returned in constant time

For example, in NWGraph, the `graph` concept is defined as follows:

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

`concept` `graph` requires that, for an algorithm to operate on a graph type G, two things needs to be defined minimally: specialization of an associated type `vertex_id_t` in G and a control point object `num_vertices` to obtain the total number of vertices in graph g.

One of the most important concepts in NWGraph, **`concept`** `adjacency_graph`, is defined as a random access range of forward ranges as follows:

```
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>>;
template <typename R>
concept vertex_list_c = std::ranges::forward_range<R> &&
  requires(std::ranges::range_value_t<R> e) {
  std::get<0>(e);
};
template <typename G>
concept adjacency_graph = graph<G>
  && std::ranges::random_access_range<G>
  && vertex_list_c<inner_range<G>>
  && requires(G g, inner_value<G> e, vertex_id_t<G> u) {
      { g[u] } -> std::same_as<inner_range<G>>;
      { std::get<0>(e) } ->  std::convertible_to<vertex_id_t<G>>;
};
```

3.3 Graph Range Adaptors

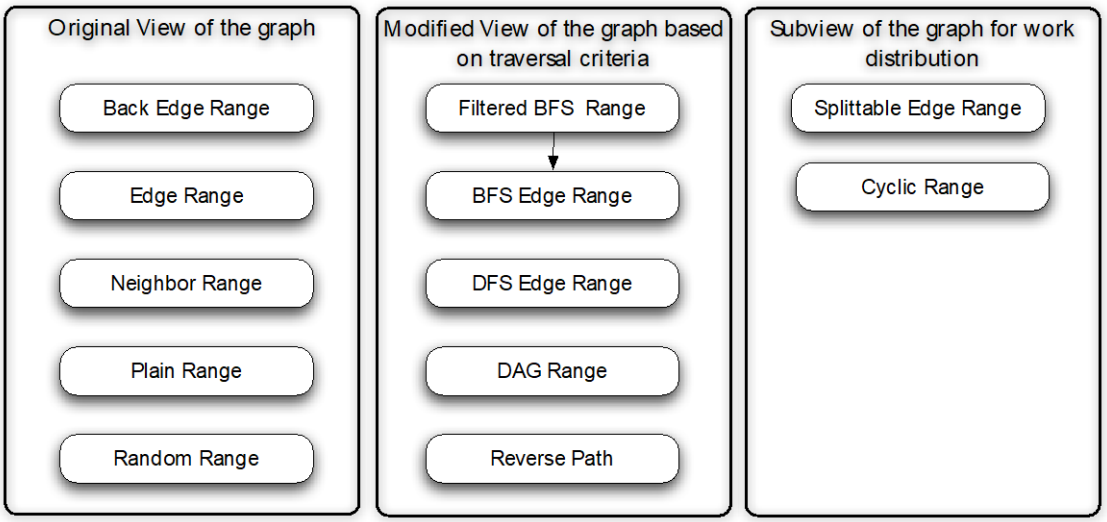
A key feature of the new C++ Ranges is the notion of views, which allow for different ways to view a range without changing the underlying data. Between a range and a range view sits a range adaptor, which takes the original range and presents it to the user as a view while hiding the underlying data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

For example, consider a breadth-first search traversal (BFS). BFS is considered a core graph algorithm kernel for performance benchmarking, but a standalone BFS traversal is rarely useful. Typically an algorithm written on top of a BFS search is more interested in doing actual computation (calculating distance from the source, finding parent list etc., for example) in the order BFS visits edges and vertices. Users are less interested in maintaining internal data structures such as queues and colormaps required to perform the traversal. An ideal abstraction will make such implementation details oblivious to the user, and instead expose only the visited edges and vertices. Range adapters are well-suited to this task, and can provide an algorithm implementor with the desired view of a graph.

```
template <typename GraphT>
void bfs_traversal(GraphT& G) {
  bfs_range bfs(G);
  for (auto&& u : bfs) {
    // Visit vertex u
  }

  bfs_edge_range bfs_edge(G);
  for (auto&& [u,v] : bfs_edge) {
    // 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. Similar to C++ range views, we can model a common set of graph range adaptors or views that can be utilized by many graph algorithms (). For example, based on the original graph, several views of the graph can be constructed. These include: Edge range, Neighbor range, Plain range, Random range and Back-edge range. Additionally, alternative view of the graph may be warranted by an algorithm. For example, BFS and DFS traversal-based algorithms consider vertices in certain order. These alternative views include: BFS edge range, Filtered BFS range, DFS edge range, DAG range and Reverse Path. We describe the graph range adaptors, their operation and applications in .



Range adaptors in NWGraph.

Range adaptors.

Range adaptor	Definition and Requirements
`edge_range`	A range adaptor that provides access to the edges in a graph. • Input: Graph g. • Return[in each iteration]: One edge (u, v) and any associated edge property.
`neighbor_range`	A range adaptor that provides access to the inner range (neighborlist of a vertex) in a graph. • Input: Graph g. • Return[in each iteration]: A vertex key and its neighborlist range.
`plain_range`	A range adaptor that provides access to the outer range (list of vertices) in a graph. • Input: Graph g. • Return[in each iteration]: A vertex key.
`random_range`	A range adaptor for a random walk of the underlying graph. • Input: Graph g, starting vertex, maximum number of steps to be taken from the start, and a seed for uniform distribution generator.

Range adaptor	Definition and Requirements
	• Return[in each iteration]: A vertex key (next neighbor to walk to).
`bfs_edge_range`	A range adaptor that provides access to edges in BFS order. • Input: Graph g, source vertex s. • Return[in each iteration]: One Edge (u, v) and any associated property (k) at a time in BFS order.
`dfs_edge_range`	A range adaptor that provides access to edges in DFS order. • Input: Graph g, source vertex s. • Return[in each iteration]: One Edge (u, v) and any associated property (k) at a time in DFS order.
`filtered_bfs_range`	A range adaptor that provides access to filtered edges in breadth-first order. • Input: Graph g, source s, target t, a lambda function to be used as a filter. • Return[in each iteration]: One edge (u, v) and the associated properties of the edge that satisfies the criteria.
`back_edge_range`	Given an edge (u, v) stores fast lookup to edge (v, u). If edge (v, u) does not exist, store it in a temporary data structure.
`reverse_path`	A range adaptor that traverses a tree from sink to source. • Input: A search tree, sink vertex and source vertex. • Return[in each iteration]: Parent vertex key of the current vertex in the tree.
`dag_range`	Directed Acyclic Graph (DAG) range is an important range adaptor that builds predecessor-successor relationships among vertices in a graph based on a given criteria (for example degree count, where max-degree vertices will be the roots of the DAG), implicitly creating a DAG view from the original graph. A DAG adaptor then traverses the graph, dictated by the predecessor-successor relationship. In each iteration, it returns an edge (v, u) from the imposed DAG and a flag `(ready_to_process)` to indicate if a successor (u) is ready to process (for example color a successor vertex only when all the predecessor vertices have been assigned a color).

Range adaptor	Definition and Requirements
	• Input: Graph g, predecessor and successor list for all vertices based on a criteria. • Return[in each iteration]: One edge (v,u) from the implicitly constructed DAG and whether the successor in the edge (u) is ready to be processed.
`splittable_range`	A range adaptor that provides a subview of the outer range of the graph. • Input: Graph g, an optional cutoff bound. • Return: a subview (a sub range) of g.
`cyclic_range`	A range adaptor that partitions the outer range of the graph in a cyclic manner based on the specified stride. • Input: Graph g, a stride integer. • Return[in each iteration]: A sub range of g.

4.0 Algorithms and Data Structures

In this section, we discuss the algorithms implemented in NWGraph in and demonstrate the applications of different NWGraph adapators to various well-known graph algorithms (Cormen et al. 2009) in .

Algorithms in NWGraph.

Algorithm	Definition
Breadth-first search	Traverses a graph in breadth-fist search order from a given source. Implementation includes: top-down, bottom-up and direction-optimized (Beamer, Asanović, and Patterson 2012) 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 (Meyer and Sanders 2003) is implemented.
Connected component	Finds connected components in a graph. Implementations include Afforest (Sutton, Ben-Nun, and Barak 2018), Shiloach-Vishkin (Shiloach and Vishkin 1980), BFS-based (J. Shun, Dhulipala, and Blelloch 2014) and minimal label propagation (Orzan 2004; Yan et al. 2014) algorithms.
Page rank	Compute the importance of each vertex in a graph. Implements the Gauss-Seidel algorithm (Arasu et al. 2002).
Triangle counting	Counts the number of triangles in a graph. Implements algorithms discussed in (Lumsdaine et al. 2020).
Betweenness centrality	Measures how many times each vertex lies on the shortest paths to other vertices. Brandes Algorithm (Brandes 2001) has been implemented.
Maximum flow	Given a source and a sink, find paths with available capacity and to push flow through them until no more paths are available. Implements Edmonds-Karp algorithm.
K-core	Finds the subgraph induced by removing all vertices with degreeless 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 (Jones and Plassmann 1993).
Maximal independent set	Graph coloring with two colors.

Application of Range adaptors in different algorithms.

Range adaptor	Application to algorithm
`edge_range`	K-core computation, triangle counting, Jaccard similarity, finding connected components, page rank, sparse matrix-vector multiplication (SpMv), graph coloring, DAG-based Maximal Independent set.
`neighbor_range`	Breadth-first search, triangle counting, finding connected components, sparse matrix-vector multiplication (SpMv).
`plain_range`	Finding connected components.
`random_range`	Random walk.
`bfs_edge_range`	Any algorithm that leverages Breadth-first search traversal (for example, calculating the distance of vertices from the source, finding the predecessor list of vertices in BFS order etc.)
`dfs_edge_range`	Similar to the BFS edge range except done in the Depth-first search order.
`filtered_bfs_range`	Maximum flow
`back_edge_range`	Maximum flow
`reverse_path`	To find the predecessor list on a traversed path, for example: to compute allowed flow in maximum flow, Breadth-first search.
`dag_range`	Graph coloring, maximal independent set.
`splittable_range`	To provide a subview of the graph, either to access a subgraph, or to partition the graph among multiple threads.
`cyclic_range`	To distribute the graph among multiple threads in a cyclic manner to achieve load balancing for certain graph inputs.

4.1 Data Structures

While a key goal of the NWGraph library is generic graph algorithms that work with a variety of user defined containers, it would be impossible to provide any proof of concepts without our own graph types. Furthermore, some users of the library might not have their own containers and would make use of a variety of built in container that satisfy different concept requirements. Considering these, the following set of data structures are available in NWGraph ().

Data structures in NWGraph.

Data Structure	Description
`struct_of_array`	**`struct`** of arrays (`tuple` of `vector`s)
`array_of_struct`	Array of **`struct`**s (`vector` of `tuple`s)

Data Structure	Description
`indexed_struct_of_array`	A indexed **`struct`** of array, where, with a vertex id, the neighborlist of the vertex can be accessed. In other words, the outer range can be indexed to retrieve the inner range (neighborlist).
`edge_list`	a `vector` of edges
`vector_of_vector`	A `vector` of `vector`s, with each vector containing the neighborlist.
`vector_of_list`	A `vector` of `list`s, with each list containing the neighborlist.
`vector_of_forward_list`	A `vector` of `forward_list`s, with each forward list containing the neighborlist.
`adjacency`	This is an implementation of the Compressed Sparse Row (CSR) data structure, where two arrays are maintained (assuming contiguous vertex ids): one contains the prefix sum of the number of neighbors, and the second array maintains the list of neighbors compactly.
`array_of_list_of_struct`	An array of `list` of **`struct`**s

5.0 Extended Example

To illustrate the effectiveness of NWGraph's "there is no graph" design philosophy, this section walks through an example that ingests a database table and uses graph algorithms for some basic knowledge discovery. The desired goal is to the find the popularized Bacon number, which is the degree of separation of various Hollywood actors from the actor Kevin Bacon. The starting point of this example is an Oracle web crawl of Wikipedia which is then parsed by an open source script to create a Wikipedia movie database, with several entries of the following form.

```
{
  "title": "Movie Title",
  "cast": "Actor1","Actor2",
  "directors": "Director",
  ...
  "year": year
}
```

This is not a graph; it is a relational database that we wish to query with graph algorithms. After reading into a vector of json entries, we will populate an NWGraph edge list with directed edges from movie titles to actors. We will also keep a map from movie titles to movie ids, and from actor names to actor ids.

```
std::vector<json> jsons;
// Populate jsons vector

std::map<std::string, size_t> titles_map, names_map;
std::vector<std::string>      titles, names;
nw::graph::edge_list<nw::graph::directed> edges;
for (auto& j: jsons) {
  auto title = j["title"]
  if (titles_map.find(title) == titles_map.end()) {
    // Add title to title map if it doesn't exist
  }
  // Movie has multiple cast members
  for (auto& k : j["cast"]) {
    auto name = delink(k);
    if (names_map.find(name) == names_map.end()) {
      // Add actor to actor map if it doesn't exist
    }
    // Add title->actor edge to edge list
    edges.push_back(titles_map[title], names_map[name]);
}}
```

It will be useful to have adjacency access to both the actors involved in a movie $Adj[movie]$ and the movies and actor is involved in $Adj[actor]$, so we create adjacency structures for both sets. Note that since these sets to not overlap, we really have a bipartite graph.

```
// Adj[movie], from title to actor
auto G = nw::graph::adjacency<0>(edges);
// Adj[actor], from actor to title
auto H = nw::graph::adjacency<1>(edges);
```

Two actors collaborate a movie if there is a length-two path between them in this bipartite graph. To make this a direct relationship, we construct an edge list which connects actors if this relationship exists. We will need adjacency lookup into this collaboration graph, so we then compress it to the final costar adjacency structure.

```
nw::graph::edge_list<nw::graph::undirected, size_t> collaborations;
for (size_t actor = 0; actor < H.size(); ++actor) {//for each actor
  for (auto&& [movie] : H[actor]) {//for each movie of actor
    for (auto&& [another_actor] : G[movie]) {//for another_actor of movie
      if (actor != another_actor) {//exclude the actor self
        collaborations.push_back(actor, another_actor, movie);
}}}}
auto costar = nw::graph::adjacency<0, size_t>(collaborations);
```

Now we can perform a breadth first search from Kevin Bacon to find the Bacon number of every actor. This makes use of the bfs_edge_range adaptor to provide a lightweight view of the BFS traversal.

```
size_t bacon = names_map["Kevin Bacon"];
std::vector<size_t> bacon_number(costar.size());
std::vector<size_t> parents(costar.size());
std::vector<size_t> together_in(costar.size());
for (auto&& [u, v, k] :
  nw::graph::bfs_edge_range(costar, bacon)) {
  bacon_number[v] = bacon_number[u] + 1;
  parents[v]      = u;
  together_in[v]  = k;
}
```

After writing a path_to_bacon helper function to traverse the parent tree, we can query the Bacon number of an actor to find their relationship to Kevin Bacon. For example, we can check the Bacon numbers of the actors that have played *Batman*, and find that none of them have a Bacon number greater than two.

```
path_to_bacon("Adam West");
path_to_bacon("Michael Keaton");
% path_to_bacon("Val Kilmer");
% path_to_bacon("George Clooney");
% path_to_bacon("Christian Bale");
% path_to_bacon("Ben Affleck");
% path_to_bacon("Robert Pattinson");

/* Output
Adam West has a Bacon number of 2
  Adam West with Frank Welker
  in {Aloha, Scooby-Doo!}
```

```
  Frank Welker with Kevin Bacon
  in {Balto (film)}
Michael Keaton has a Bacon number of 1
  Michael Keaton with Kevin Bacon
  in {She's Having a Baby}
*/
```

6.0 Evaluation

In this section, we discuss our experimental results. First we evaluate the abstraction penalty for different ways of iterating over a graph as ranges of ranges. Next we evaluate the abstraction penalty for representing a graph with different data structures. Finally, we compare the performance of our NWGraph library with 3 other well-known graph frameworks: GAP graph benchmark suite (Y. Zhang et al. 2018), Galois (Kulkarni et al. 2007) graph library and GraphIt (Y. Zhang et al. 2018) domain-specific language for graphs.

6.1 Experimental Setup

Abstraction penalty benchmarks were run on 2019 MacBook Pro with 2.4 GHz 8-Core Intel® Core i9 processor with 64 GB DDR4 memory running at 2.6GHz. All performance measurements were collected on Intel® Xeon® -based servers. Each server contains two Intel® Xeon® Platinum 8153 processors, each with 16 physical cores (32 logical cores) running at 2.0 GHz. Each processor has 22 MB L3 cache. The total system memory of each server is 384 GB DDR4 running at 2.6 GHz.

6.2 Abstraction Penalty

While ranges and range based for loops are useful programming abstractions, it is important to consider any performance abstraction penalties associated with their use. We benchmark these penalties to ensure they will not significantly limit performance compared to "raw for loop" implementation. For example let us consider the sparse matrix-dense vector multiplication (SpMV) kernel used in page rank, which multiplies the adjacency matrix representation of a graph by a dense vector x and stores the result in another vector y. Using a compressed sparse row (CSR) data structure to store the adjacency matrix, a raw for loop implementation would access the indices and weights of edges with pointers into the CSR data structure.

```
// Raw for loop SpMV
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];
 }
}
```

However NWGraph does not assume this underlying CSR structure, and would prefer to write these algorithms more generically with iterator based or range based for loops shown below. Note that the previous raw loop implementation had access to information that the SpMV kernel does not actually need, which is random access into a vertex's incidence list. The incidence list only needs to be traversed in some order to produce the desired result.

```
// Iterator based for loop SpMV
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;
}
// Range based for loop SpMV
vertex_id_t k = 0;
for(auto&& i : G){
 for(auto&& [j, v] : i){
  y[k] += x[j] * v;
 }
 ++k;
}
```

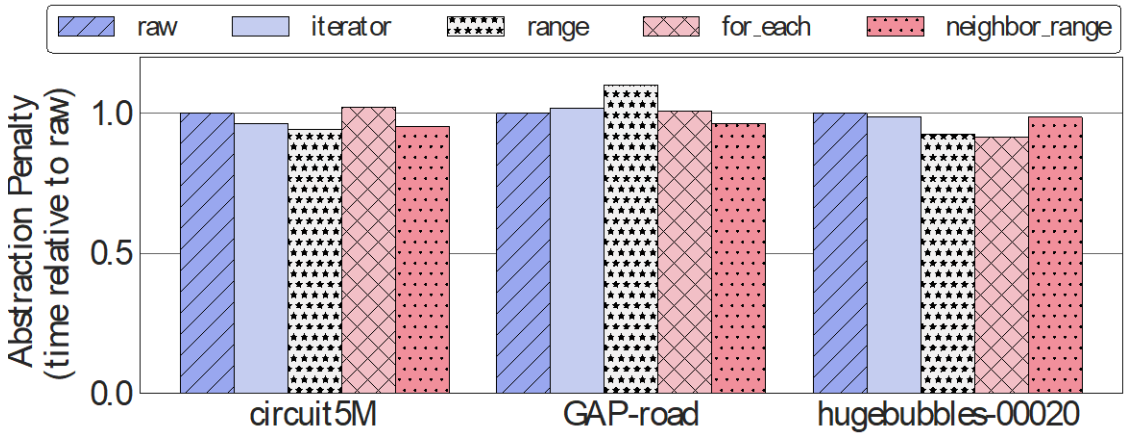
Iterators can also be used to process the edges with std::for_each.

```
// STL for_each SpMV
std::for_each(G.begin(), G.end(), [&](auto&& e) {
  y[std::get<0>(e)] += x[std::get<1>(e)] * std::get<2>(e);
});
```

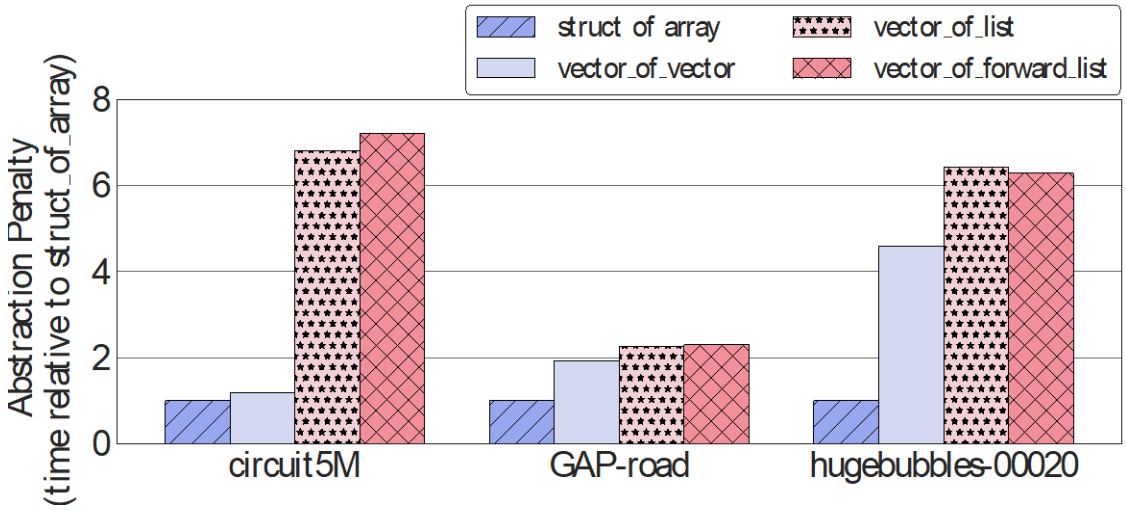
Finally, with one of our range adaptors, `neighbor_range`, we can easily access the indices and the neighbors of it.

```
// Neighbor range based for loop (SpMV)
for (auto&& [i, neighbors] : neighbor_range(G)) {
  for (auto&& [j, v] : neighbors) {
    y[i] += x[j] * v;
  }
}
```

There are even more combinations of these traversals that are omitted for lack of space (combinations of ranges and iterators, with and without compound initializers, auto vs auto&& etc.)



Different data access abstractions (`iterator`s, `range`s, `std::for_each` and `neighbor_range` adaptor) with their abstraction penalties measured relative to a raw for loop implementation. There is no significant performance penalty relative to the raw loop implementation.



*Measured abstraction penalty for the SpMV benchmark with graphs represented by different containers using `iterator` based **`for`** loop. The execution time has been normalized w.r.t to the execution time of SpMV with graphs represented as `struct_of_array` (lower is better).*

To experimentally evaluate the abstraction penalty, we consider SpMV with three graphs with different underlying topologies from the SuiteSparse matrix collection: circuit5M, GAP-road, and hugebubbles. These were chosen because they have similar numbers of edges (30M to 60M) and the benchmarks run in comparable time. Timing results were averaged over 5 runs of each benchmark. shows the results of the different data access abstractions relative to the raw loop timing, for 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 penalty relative to the raw loop implementation.

We also evaluated the abstraction penalty incurred for storing a graph in different containers. In particular, we have selected `struct_of_array`, `vector_of_vector`, `vector_of_list`, `vector_of_forward_list` containers. Note that, all these containers meet the requirement of our `graph` concept. We consider SpMV benchmark implemented with iterator based for loop with circuit5M, GAP-road, and hugebubbles datasets. shows the performance of SpMV with different containers. The execution time is normalized relative to the execution time of SpMV with `struct_of_array` container. As can be observed from , SpMV with `struct_of_array` performs best, followed by `vector_of_vector`. `struct_of_array` representation is cache-friendly and supports random access of the outer and inner range efficiently.

6.3 Performance

In this section, we evaluate and compare the performance of NWGraph with three well-known graph frameworks: GAP, Galois and GraphIt. The evaluation is intended to assess the performance of various parallel graph algorithms available in NWGraph in the context of other HPC graph frameworks. This paper focuses on the interfaces and design decisions around the range abstraction and concepts for graphs. Hence, we do not discuss the parallelization aspects of the algorithms in details here. However, overall, NWGraph leverages Intel's Threading Building Block (TBB) (Intel (2020)) concurrent data structures for maintaining the internal states of different graph algorithms. In addition, for workload distribution among the threads, NWGraph can either use `block` range from TBB, our customized `cyclic` range adaptor, or C++ parallel execution policy (`std::execution::par, std::execution::par_unseq`) whenever appropriate. All experiments are conducted on 32 physical cores (32 threads).

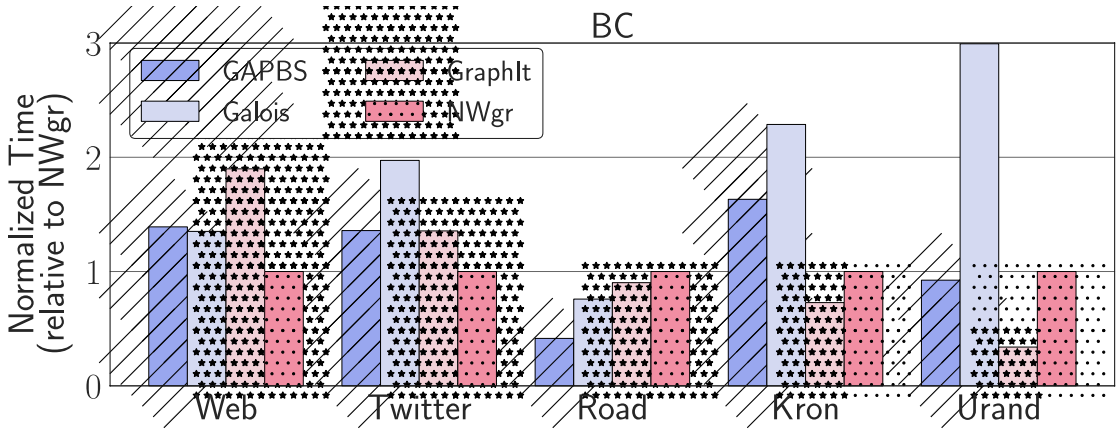
For our experiments, we chose five representative datasets according to the GAP benchmark suite (Beamer, Asanović, and Patterson (2015)). These datasets () have diverse structural properties and have been collected from various application domains. We select six different graph algorithms (Betweenness Centrality, Breadth-first Search, Connected Components, Page Rank, Single Source Shortest Path, and Triangle Counting) that are common across these graph frameworks.

We report the performance of different graph frameworks in . We summarize our observations as follows:

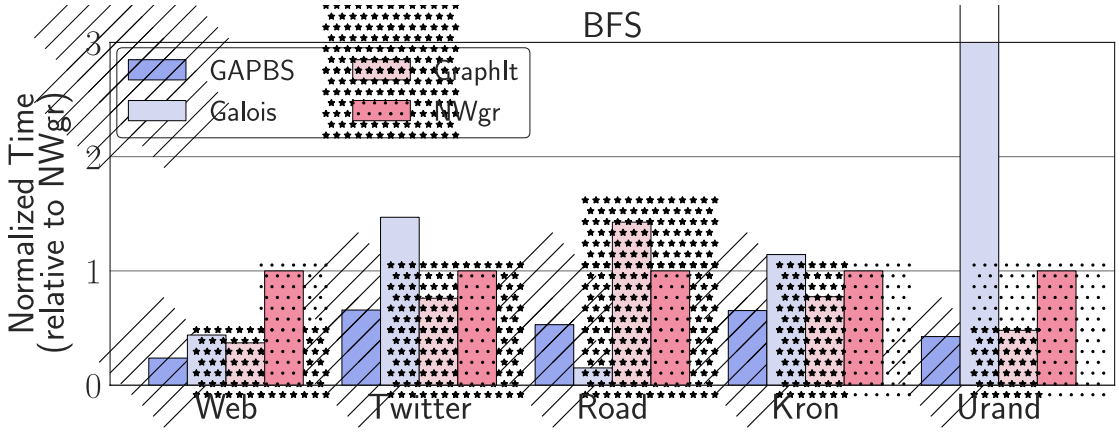
- With Web and kron datasets, which have skewed degree distribution, our triangle counting (TC) and page rank algorithms in NWGraph outperform other frameworks. Except for the road network input, for both of these graph problems, NWGraph performs comparably with other inputs.
- NWGraph also runs faster with Web, Twitter and Kron datasets (power-law graphs) for Betweenness centrality (BC) algorithms.
- NWGraph suffers performance with bounded graphs such as road network input for SSSP.
- Overall, NWGraph performs better or comparable to other graph frameworks.
- For connected component (CC), all frameworks except GraphIt implement Afforest algorithm. Hence GraphIt performs worse with all inputs for CC.

Dataset for performance evaluation

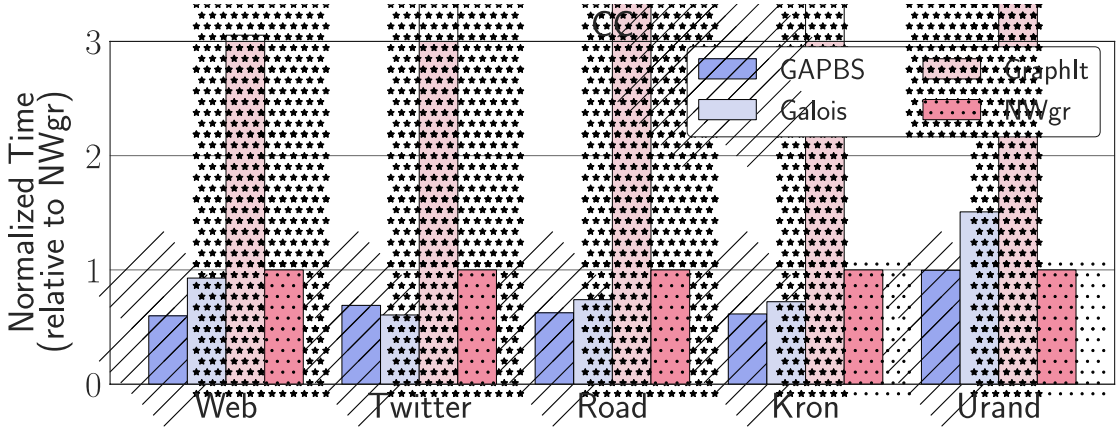
Name	Description	# Vertices (M)	# Edges (M)	Degree Distribution	References
Road	USA road network	23.9	57.7	bounded	(“9th DIMACS Implementation Challenge - Shortest Paths.” 2006)
Twitter	Twitter follower Links	61.6	1,468.4	power	(Kwak et al. 2010)
Web	Web Crawl of .sk Domain	50.6	1,930.3	power	(Boldi and Vigna 2004)
Kron	Synthetic Graph	134.2	2,111.6	power	(Murphy et al. 2010)
Urand	Uniform Random Graph	134.2	2,147.5	normal	(Erdős and Rényi 1959)



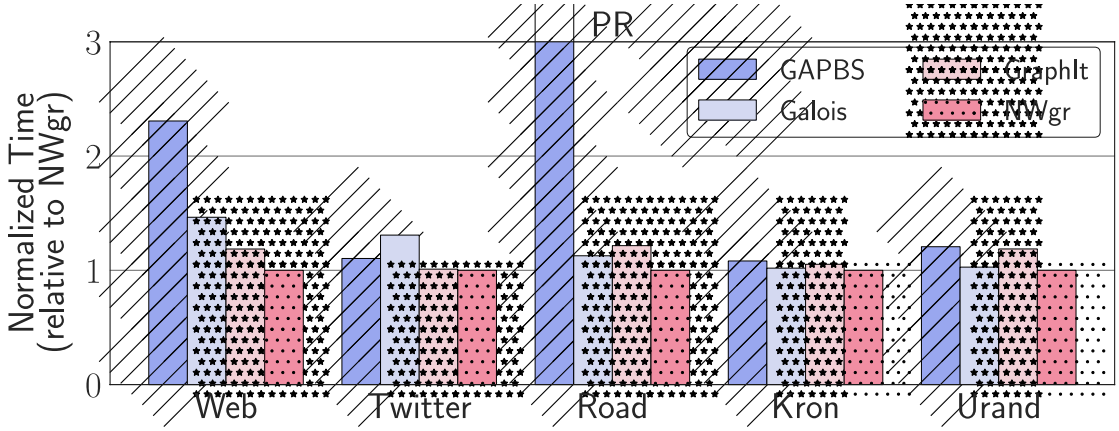
[fig:bc_perf]



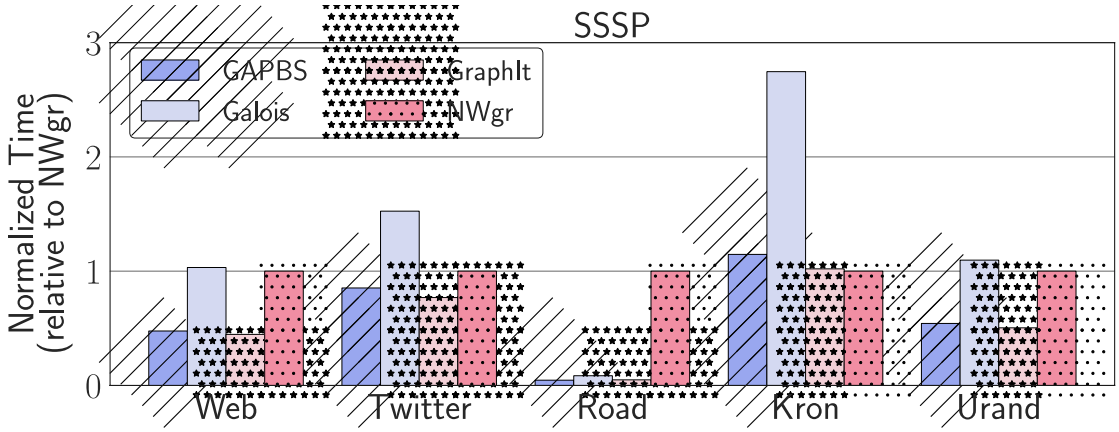
[fig:bfs_perf]



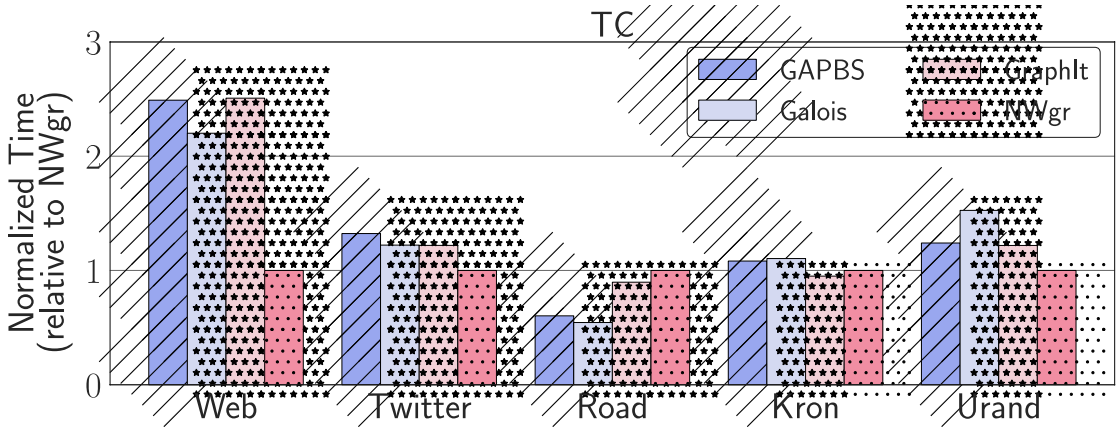
[fig:cc_perf]



[fig:pr_perf]



[fig:sssp_perf]


TC
Normalized Time
(relative to NWgr)
3
2
1
0
GAPBS
Galois
GraphIt
NWgr
Web
Twitter
Road
Kron
Urand

[fig:tc_perf]

7.0 Related Graph 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.

7.1 Generic C++ Graph Libraries

The Boost Graph Library (BGL) (Siek, Lee, and Lumsdaine 2002) and LEMON graph library (Dezső, Jüttner, and Kovács 2011) both contributed to the development of generic graph algorithms in C++. BGL proposed algorithm templates that could be used on a variety of underlying graph types, e.g. vector of vectors, vector of lists, 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, for instance recording the timestamp a vertex is touched in a BFS search. The LEMON graph library shared many of these features, and also proposed the use of graph adaptors which would allow graph algorithms to run on temporarily modified versions of a graph. For instance a subgraph adaptor would allow an algorithm to operate on a graph with some vertices and edges temporarily filtered out, with the original graph storage remaining intact, preventing expensive copies. LEMON also introduced some simple graph concepts, which user defined graphs could conform to, but which predated the official C++ Concepts. 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 which limited their use. The visitor objects are difficult to use, both from a programming and algorithmic design perspective. Property maps are a convenient programming abstraction, but can 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. As mentioned in the previous section, a major shortcoming of these designs is the difficulty of using custom data structures. In order to adapt an existing user-defined data structure, BGL requires overloading several global free functions required by the BGL interface. 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 feature in C++ such as constexpr, variadic templates, automatic type deduction, execution policies, etc.

7.2 Linear Algebra Based Graph Libraries

Many graph algorithms can be reformulated as a series of operations on sparse matrices. Several researchers are working on set of sparse linear algebra kernels for implementing graph algorithms, known as the GraphBLAS, inspired by the success of the Basic Linear Algebra Subroutine (BLAS) kernels for dense matrix operations,. A C reference implementation of GraphBLAS is available in the SuiteSparse software

package, with accompanying graph implementations available in the LAGraph library (Mattson et al. 2019). A C++ implementation known as the GraphBLAS Tempalte Library (GBTL) (P. Zhang et al. 2016) is also under development.

The goal of GraphBLAS (Kepner et al. 2016) is to standardize the necessary sparse matrix kernels for graph algorithms so that hardware and software library vendors can optimize performance of these kernels for various architectures. With some upfront cost of translating a graph algorithm to linear algebra, GraphBLAS promises scalability from a laptop to a compute cluster. It is an open question whether or not all graph algorithms can be written using a linear algebraic formulation. Furthermore, it is unclear if there are enough programmers trained to write graph algorithms with sparse matrix operations to take advantage of these kernels.

7.3 NetworKit

Inspired by the NetworkX package (Hagberg, Schult, and Swart 2008) in Python for network graph analytic, NetworKit aims to keep the user-friendly Python interface, while improving performance with algorithms written in C++ (Staudt, Sazonovs, and Meyerhenke 2016). NetworKit targets the network science domain, and provides several centrality and clustering algorithms, along with several graph generators. The cited design goals stress the composability of the existing algorithms within the larger Python ecosystem, while not much is said about the extensibility of the underlying set of algorithms.

7.4 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) (Gregor and Lumsdaine 2005), Galois (Kulkarni et al. 2007), Ligra (Julian Shun and Blelloch 2013), Giraph (Shaposhnik, Martella, and Logothetis 2015), Gunrock (Wang et al. 2016), GraphIt (Y. Zhang et al. 2020), etc. The contributions of these frameworks are typically a computational model for parallel processing of graphs, with less emphasis on the usability or extensibility of graph algorithms or containers. A through evaluation of several well-known parallel graph frameworks can be found in (Azad et al. 2020).

8.0 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 state of the art. The NWGraph library is available as open source on github. We intend to propose the design to the C++ standardization committee for consideration as a standard C++ graph library.

9.0 References

"9th DIMACS Implementation Challenge - Shortest Paths." 2006. http://www.dis.uniroma1.it/challenge9/.

Arasu, Arvind, Jasmine Novak, Andrew Tomkins, and John Tomlin. 2002. "PageRank Computation and the Structure of the Web: Experiments and Algorithms." In *WWW*, 107–17.

Azad, Ariful, Mohsen Mahmoudi Aznaveh, Scott Beamer, Mark Blanco, Jinhao Chen, Luke D'Alessandro, Roshan Dathathri, et al. 2020. "Evaluation of Graph Analytics Frameworks Using the GAP Benchmark Suite." In *2020 IEEE International Symposium on Workload Characterization (IISWC)*, 216–27. IEEE.

Beamer, Scott, Krste Asanović, and David Patterson. 2015. "The GAP Benchmark Suite." *arXiv Preprint arXiv:1508.03919*.

Beamer, Scott, Krste Asanović, and David A. Patterson. 2012. "Direction-Optimizing Breadth-First Search." *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

Boldi, Paolo, and Sebastiano Vigna. 2004. "The WebGraph Framework I: Compression Techniques." *WWW*, 595–601.

Brandes, Ulrik. 2001. "A Faster Algorithm for Betweenness Centrality." *The Journal of Mathematical Sociology* 25 (2): 163–77. https://doi.org/10.1080/0022250X.2001.9990249.

Cormen, Thomas H, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to Algorithms*. MIT press.

Dezső, Balázs, Alpár Jüttner, and Péter Kovács. 2011. "LEMON–an Open Source c++ Graph Template Library." *Electronic Notes in Theoretical Computer Science* 264 (5): 23–45.

Erdős, Paul, and Alfréd Rényi. 1959. "On Random Graphs. I." *Publicationes Mathematicae* 6: 290–97.

Gregor, Douglas, and Andrew Lumsdaine. 2005. "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*, 423–37. OOPSLA '05. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/1094811.1094844.

Hagberg, Aric A., Daniel A. Schult, and Pieter J. Swart. 2008. "Exploring Network Structure, Dynamics, and Function Using NetworkX." In *Proceedings of the 7th Python in Science Conference*, edited by Gaël Varoquaux, Travis Vaught, and Jarrod Millman, 11–15. Pasadena, CA USA.

Intel. 2020. “Intel Threading Building Blocks (TBB).” 2020. https://github.com/oneapi-src/oneTBB.

Jones, Mark T, and Paul E Plassmann. 1993. “A Parallel Graph Coloring Heuristic.” *SIAM Journal on Scientific Computing* 14 (3): 654–69.

Kepner, Jeremy, Peter Aaltonen, David Bader, Aydın Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, et al. 2016. “Mathematical Foundations of the GraphBLAS.” In *HPEC*. IEEE.

Kulkarni, Milind, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. 2007. “Optimistic Parallelism Requires Abstractions.” In *PLDI*, 211–22. ACM. https://doi.org/10.1145/1250734.1250759.

Kwak, Haewoon, Changhyun Lee, Hosung Park, and Sue Moon. 2010. “What Is Twitter, a Social Network or a News Media?” *WWW*.

Lumsdaine, Andrew, Luke Dalessandro, Kevin Deweese, Jesun Firoz, and Scott McMillan. 2020. “Triangle Counting with Cyclic Distributions.” In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, 1–8. https://doi.org/10.1109/HPEC43674.2020.9286220.

Mattson, Tim, Timothy A. Davis, Manoj Kumar, Aydin Buluc, Scott McMillan, José Moreira, and Carl Yang. 2019. “LAGraph: A Community Effort to Collect Graph Algorithms Built on Top of the GraphBLAS.” In *GrAPL at IPDPS*, 276–84. IEEE.

Meyer, Ulrich, and Peter Sanders. 2003. “Δ-Stepping: A Parallelizable Shortest Path Algorithm.” *Journal of Algorithms* 49 (1): 114–52. https://doi.org/https://doi.org/10.1016/S0196-6774(03)00076-2.

Murphy, Richard C., Kyle B. Wheeler, Brian W Barrett, and James A. Ang. 2010. “Introducing the Graph 500.” In *Cray User’s Group*. CUG.

Musser, David R., and Alexander A. Stepanov. 1989. “Generic Programming.” In *International Symposium ISSAC 1988*, edited by P Gianni, 38:13–25. Lecture Notes in Computer Science. Springer-Verlag.

Niebler, Eric, Casey Carter, and Christopher Di Bella. 2018. “The One Ranges Proposal.” Tech. rep. P0896r4. Nov. 2018. url: http://www. open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0896r4. pdf.

Orzan, S. M. 2004. “On Distributed Verification and Verified Distribution.” Ph.D. thesis, VRIJE UNIVERSITEIT. http://dare.ubvu.vu.nl/handle/1871/10338.

Shaposhnik, Roman, Claudio Martella, and Dionysios Logothetis. 2015. *Practical Graph Analytics with Apache Giraph*. 1st ed. edition. New York: Apress.

Shiloach, Yossi, and Uzi Vishkin. 1980. “An O(log n) Parallel Connectivity Algorithm.” Computer Science Department, Technion.

Shun, J., L. Dhulipala, and G. Blelloch. 2014. “A Simple and Practical Linear-Work Parallel Algorithm for Connectivity.” In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*, 143–53. SPAA ’14. New York, NY, USA: ACM. https://doi.org/10.1145/2612669.2612692.

Shun, Julian, and Guy E. Blelloch. 2013. “Ligra: A Lightweight Graph Processing Framework for Shared Memory.” In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 135–46. PPoPP ’13. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/2442516.2442530.

Siek, Jeremy G., Lie-Quan Lee, and Andrew Lumsdaine. 2002. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley.

Staudt, Christian L., Aleksejs Sazonovs, and Henning Meyerhenke. 2016. “NetworKit: A Tool Suite for Large-Scale Complex Network Analysis.” *Network Science* 4 (4): 508–30. https://doi.org/10.1017/nws.2016.20.

Stepanov, Alexander, and Meng Lee. 1995. “The Standard Template Library.” HPL-95-11. HP Laboratories.

Sutton, Michael, Tal Ben-Nun, and Amnon Barak. 2018. “Optimizing Parallel Graph Connectivity Computation via Subgraph Sampling.” In *IPDPS*, 12–21. IEEE.

Wang, Yangzihao, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2016. “Gunrock: A High-Performance Graph Processing Library on the GPU.” In *PPoPP*.

Yan, Da, James Cheng, Kai Xing, Yi Lu, Wilfred Ng, and Yingyi Bu. 2014. “Pregel Algorithms for Graph Connectivity Problems with Performance Guarantees.” *Proc. VLDB Endow.* 7 (14): 1821–32. https://doi.org/10.14778/2733085.2733089.

Zhang, Peter, Marcin Zalewski, Andrew Lumsdaine, Samantha Misurda, and Scott McMillan. 2016. “GBTL-CUDA: Graph Algorithms and Primitives for GPUs.” In *GABB at IPDPS*, 912–20. IEEE.

Zhang, Yunming, Ajay Brahmakshatriya, Xinyi Chen, Laxman Dhulipala, Shoaib Kamil, Saman Amarasinghe, and Julian Shun. 2020. “Optimizing Ordered Graph Algorithms with GraphIt.” In *CGO*, 158–70. ACM. https://doi.org/10.1145/3368826.3377909.

Zhang, Yunming, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. “GraphIt: A High-Performance Graph DSL.” *PACMPL/OOPSLA* 2 (October): 121:1–30.

Pacific Northwest National Laboratory

902 Battelle Boulevard
P.O. Box 999
Richland, WA 99354
1-888-375-PNNL (7665)

www.pnnl.gov