Generic Graph Libraries in C++20

ANDREW LUMSDAINE, PHIL RATZLOFF
About Us

• Andrew Lumsdaine
  • Principal Software Engineer, TileDB, Inc and Affiliate Professor, Paul G Allen School of Computer Science and Engineering, University of Washington
  • Andrew has worked in many areas related to high-performance computing, including systems, programming languages, software libraries, and large-scale graph analytics. Open-source software projects resulting from his work include the Matrix Template Library, the Boost Graph Library, and Open MPI.

• Phil Ratzloff
  • Distinguished Software Developer, SAS Institute
  • Phil is a Distinguished Software Developer and C++ advocate at SAS Institute. He has used C++ for 26 years on applications using graphs for business cost analysis and fraud detection.

• Thanks also to Jesun Firoz, Tony Liu, Scott McMillan, Haley Riggs
Acknowledgments and Disclaimers

• Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation, TileDB, Inc., SAS Institute, Pacific Northwest National Laboratory, U.S. Department of Energy, the University of Washington, or anyone else.
Graphs Are Fundamental Abstractions

• Graphs model relationships between elements of a data set
• Without regard to what the data set actually is
• Graph theoretical (abstract) results can be applied to many different practical (concrete) problems – theory reuse
• Goes hand-in-glove with goals of generic software libraries
Graphs Are Ubiquitous

You almost surely used graph algorithms today

It is critical to have reusable software libraries to realize the reusable theories
Basic Principles

• The C++ standard library (nee STL) provides a rich set of “one-dimensional” algorithms and data structures (but not graphs)
• And standardized mechanisms for defining the type requirements that form the interfaces to generic algorithms (codified as concepts)
• Our claim: The standard library already provides sufficient capability to support generic graph algorithms and data structures
• Generic graph algorithms can be defined with range of ranges as the type requirement on input (graph) types
• Compositions of standard library containers meet these requirements
• Other graph structures can provide the required interface (just as a third-party container can provide a library-compliant interface)
Desiderata for a Graph Library

- Graphs are not for storing data, but rather for efficient algorithmic traversal of structure implied by relationships among (and of) data
- Embrace modern C++ idioms (programming practice plus, e.g., concepts, ranges, CPOs)
- Embrace modern C++ standard library
- Embrace scale of real world graphs (billions of vertices / edges)
- Prefer elegance and usability over expert-friendliness
- Genericity: Abstract from concrete, efficient algorithms to obtain generic algorithms that can be combined with different data representations to produce a wide variety of useful software (Musser and Stepanov)
Overview

• Introduction and Overview
• Review of Generic Programming and Graph Terminology
• **Requirements Analysis: Algorithms, Types, and Concepts**
• Graph Adaptors
• Concrete Data Structures
• (Extended Example: Six Degrees of Kevin Bacon)
• Towards Standardization
• Lessons Learned
Generic Programming and Generic Libraries

• Generic programming is a sub-discipline of computer science that deals with finding abstract representations of efficient algorithms, data structures, and other software concepts, and with their systematic organization.

• The goal of generic programming is to express algorithms and data structures in a broadly adaptable, interoperable form that allows their direct use in software construction.

[M. Jazayeri, R. Loos, D. Musser, and A. Stepanov, 1998]
Generic Programming Methodology

1. Study the concrete implementations of an algorithm

2. Lift away unnecessary requirements to produce a more abstract algorithm
   a) Catalog these requirements
   b) Bundle requirements into concepts

3. Repeat the lifting process until we have obtained a generic algorithm that:
   a) Captures the essence of the “higher truth” of that algorithm
   b) Instantiates to efficient concrete implementations
STL Architecture

<table>
<thead>
<tr>
<th>Container Classes</th>
<th>Iterators</th>
<th>Generic Algorithms</th>
<th>Function Objects</th>
<th>Adaptors</th>
</tr>
</thead>
<tbody>
<tr>
<td>vector T</td>
<td></td>
<td>sort</td>
<td>less T</td>
<td>stack C</td>
</tr>
<tr>
<td>list T</td>
<td></td>
<td>find</td>
<td>equal T</td>
<td>queue C</td>
</tr>
<tr>
<td>istream T</td>
<td>istream_iterator T</td>
<td>merge</td>
<td>greater T</td>
<td>reverse T</td>
</tr>
<tr>
<td>ostream T</td>
<td>ostream_iterator T</td>
<td></td>
<td></td>
<td>iterator</td>
</tr>
</tbody>
</table>

++ **Increment**  ==, & **Compare, Reference**
= **Assign**  -- **Decrement**
* **Dereference**  +, -, < **Random Access**

Generic Parameter
Lifting Summation

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

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

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

double sum(node* first, node* last) {
    double s = 0;
    while (first != last) {
        s = s + first->data;
        first = first->next;
    }
    return s;
}
```
template <class InputIterator>
value_type sum(InputIterator first, 
    InputIterator last) {
    value_type s = 0;
    while (first != last)
        s = s + *first++;
    return s;
}
Lifting Summation

```c
float product(node* first, node* last) {
    float s = 1;
    while (first != last) {
        s = s * first->data;
        first = first->next;
    }
    return s;
}
```
Generic Accumulate

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

- Generic form captures all accumulation:
  - Any kind of data (int, float, string)
  - Any kind of sequence (array, list, file, network)
  - Any operation (add, multiply, concatenate)
- Instantiates to efficient, concrete implementations

Accumulate a range of T
(requires: iterator range, op(T + T))

Sum of an range of T
(requires: iterator range, T + T)

Sum of an array of T
(requires: T + T)

Sum of an array of integers

Sum of an array of floats

Sum of a list of floats

Product of a list of integers
Lifting and Specialization

- Specialization is **dual** to lifting
- Synthesizes efficient code for a particular use of a generic algorithm:

```cpp
int array[20];
accumulate(array, array + 20, 0, std::plus<int>());
```

- ... generates the **same code** as our initial sum function for integer arrays.
Let’s Apply the Generic Programming Process to Graphs
This is what is important

$G = \{V, E\}$

$V = \{\text{GND}, \text{Vdd}, n0, n1, n2, \text{Vout}\}$

$E = \{(n0, n1), (\text{Vdd}, \text{GND}), (n0, \text{Vdd}), (n2, \text{Vdd}), (\text{GND}, n2), (n2, \text{Vout}), (\text{GND}, n0), (n2, n1)\}$
This is what is important
Traversals is a fundamental operation in graph algorithms

- Given a vertex $u$, find all the neighbors of $u$ (all vertices $v$ s.t. $(u, v) \in E$, i.e., s.t. edge $(u, v)$ is in the graph)
- Then for each neighbor, find its neighbors (and so on)

$G = \{V, E\}$

$V = \{0, \text{Vdd}, n0, n1, n2, \text{Vout}\}$

$E = \{(n0, n1), (\text{Vdd}, 0), (n0, \text{Vdd}), (n2, \text{Vdd}), (0, n2), (n2, \text{Vout}), (0, n0), (n2, n1)\}$
Adjacency “List”

- An adjacency list $\text{Adj}(G)$ is an array of size $|V|$
- Each entry $\text{Adj}(G)[u]$ contains all vertices $v$ for which $(u, v)$ is in $E$
- Implication: Vertices are indexes $0, 1, \ldots |V| - 1$
- Implication: If $(u, v) = (v, u)$ then $v \in \text{Adj}(G)[u]$ and $u \in \text{Adj}(G)[v]$

$$
\begin{align*}
G &= \{V, E\} \\
V &= \{0, 1, 2, 3, 4, 5\} \\
E &= \{(1, 3), (2, 0), (1, 2), (5, 2), (0, 5), (5, 4), (0, 1), (5, 3)\}
\end{align*}
$$

Edges w/ neighbors of 5

Constant time lookup

Iterable, not necessarily a linked list
Index Graphs

• Did you notice the sleight of hand?

\[ G = \{V, E\} \]
\[ V = \{0, Vdd, n0, n1, n2, Vout\} \]
\[ E = \{(n0, n1), (Vdd, 0), (n0, Vdd), (n2, Vdd), (0, n2), (n2, Vout), (0, n0), (n2, n1)\} \]

\[ G' = \{V', E'\} \]
\[ V' = \{0, 1, 2, 3, 4, 5\} \]
\[ E' = \{(1, 3), (2, 0), (1, 2), (5, 2), (0, 5), (5, 4), (0, 1), (5, 3)\} \]
Principle: Graphs Represent Structure

Structure is in here (implicitly)

Library provided

Library provided

User shouldn't be building this manually

Index

<table>
<thead>
<tr>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>GND</td>
</tr>
<tr>
<td>Vdd</td>
</tr>
<tr>
<td>n0</td>
</tr>
<tr>
<td>n1</td>
</tr>
<tr>
<td>n2</td>
</tr>
<tr>
<td>nout</td>
</tr>
</tbody>
</table>

Index Adjacency List

<table>
<thead>
<tr>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>n0 n1 C1</td>
</tr>
<tr>
<td>Vdd GND AC</td>
</tr>
<tr>
<td>n0 Vdd R2</td>
</tr>
<tr>
<td>n2 Vdd R0</td>
</tr>
<tr>
<td>GND n2 R1</td>
</tr>
<tr>
<td>n2 Vout L1</td>
</tr>
<tr>
<td>GND n0 C0</td>
</tr>
<tr>
<td>n2 n1 R3</td>
</tr>
</tbody>
</table>

Index Edge List

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1 3</td>
</tr>
<tr>
<td>2 0</td>
</tr>
<tr>
<td>1 2</td>
</tr>
<tr>
<td>5 2</td>
</tr>
<tr>
<td>0 5</td>
</tr>
<tr>
<td>5 4</td>
</tr>
<tr>
<td>0 1</td>
</tr>
<tr>
<td>5 3</td>
</tr>
</tbody>
</table>

cf: neo4j

Algorithms use this

Structure, not data
Adjacency-List Algorithms: Breadth-First Search

- Systematically explores graph from starting vertex s
- Find all vertices reachable on an edge from s (level 1)
- Find all unvisited vertices reachable on an edge from those
- Etc

```
while (! done) {
  u = visited vertex {
    for v in neighbors(u) {
      if (v not seen) {
        visit v;
      }
    }
  }
}
```
Adjacency-List Algorithms

BFS\((G, s)\)

1. for each vertex \(u \in V(G)\)
2. \(\text{color}[u] \leftarrow \text{WHITE}\)
3. \(\text{color}[s] \leftarrow \text{GRAY}\)
4. \(Q \leftarrow \emptyset\)
5. \(\text{ENQUEUE}(Q, s)\)
6. while \(Q \neq \emptyset\)
7. \(u \leftarrow \text{DEQUEUE}(Q)\)
8. for each \(v \in Adj(G)[u]\)
9. if \(\text{color}[v] = \text{WHITE}\)
10. \(\text{color}[v] \leftarrow \text{GRAY}\)
11. \(\text{ENQUEUE}(Q, v)\)
12. \(\text{color}[u] \leftarrow \text{BLACK}\)

Enumerate vertices

Vertices can random access into containers

“Vertex properties”

Enumerate neighbor vertices

\(v \in Adj(G)[u] \rightarrow (u, v) \in E(G)\)

cf: CLRS
Minimalist Approach: Index Adjacency Graph

```cpp
template <class Graph>
auto bfs(const Graph& graph, vertex_id_t<Graph> source) {
    using vertex_id_type = vertex_id_t<Graph>

    std::vector<COLOR> color(size(graph));
    for (vertex_id_type u = 0; u < size(graph); ++u) {
        color[u] = WHITE;
    }
    color[source] = GREY;

    std::queue<vertex_id_type> Q;
    Q.push(source);

    while (!Q.empty()) {
        auto u = Q.front();
        Q.pop();
        for (auto& v : graph[u]) {  // neighbor vertex
            if (color[v] == WHITE) {
                color[v] = GREY;
                Q.push(v);
            }
        }
        color[u] = BLACK;
    }
}
```
Requirements: Basic BFS algorithms

- The graph $G$ is 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 $G$) is a *forward range*, meaning it is something that can be iterated over and have values extracted.
- The value type of the inner range is something that can be used to index into $G$ — $G[u]$ is a valid expression (returning the inner range).
- 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.
Adjacency-List Algorithms II: Dijkstra's Algorithm

- Solves single-source shortest-paths problem on a weighted, directed or undirected graph
- All edge weights must be non-negative
- Iteratively grows a set of vertices to which it knows the shortest path

\[ d[v] = \min(w(u, v) + d[u], d[v]) \]

```java
while (!done) {
    u = min distance vertex {
        for v in neighbors(u) {
            if (d[u] + weight(u, v) < d[v]) {
                d[v] = d[u] + weight(u,v);
            }
        }
    }
}
```
Adjacency-List Algorithms, II

\texttt{DIJKSTRA(G, w, s)}

1. \textbf{for} each vertex \( u \in V(G) \)
2. \hspace{1cm} \( d[u] \leftarrow \infty \)
3. \hspace{1cm} \( \pi[u] \leftarrow \text{NIL} \)
4. \( d[s] \leftarrow 0 \)
5. \( Q \leftarrow V(G) \)
6. \textbf{while} \( Q \neq \emptyset \)
7. \hspace{1cm} \( u \leftarrow \text{EXTRACT-MIN}(Q) \)
8. \hspace{1cm} \textbf{for} each \( v \in \text{Adj}(G)[u] \)
9. \hspace{2cm} \textbf{if} \( d[v] > d[u] + w(u, v) \)
10. \hspace{2cm} \( d[v] \leftarrow d[u] + w(u, v) \)
11. \hspace{2cm} \( \pi[v] = u \)

- \textit{Enumerate vertices}
- Vertices can random access into containers
- \textit{Enumerate neighbor vertices}
- \textit{Compute properties from vertex pairs (aka “edge”)}
Stored Property Graph

**V**
- GND
- Vdd
- n0
- n1
- n2
- out

**E**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>C1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td></td>
<td>AC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>2</td>
<td>R2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Edge properties**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>C0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>R2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Neighbor index**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Edge property**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Index Property Graph

\[
V = \begin{array}{ccc}
\text{GND} & \text{n0} & \text{n1} \\
\text{Vdd} & \text{Vdd} & \text{GND} \\
\text{n0} & \text{Vdd} & \text{R2} \\
\text{n2} & \text{Vdd} & \text{R0} \\
\text{n2} & \text{GND} & \text{n2} \\
\text{n2} & \text{Vout} & \text{L1} \\
\text{GND} & \text{n0} & \text{C0} \\
\text{n2} & \text{n1} & \text{R3}
\end{array}
\]

\[
E = \begin{array}{ccc}
1 & 3 & 0 \\
2 & 0 & 1 \\
1 & 2 & 2 \\
5 & 2 & 3 \\
0 & 5 & 4 \\
5 & 4 & 5 \\
0 & 1 & 6 \\
5 & 3 & 7
\end{array}
\]

Edge indices:

\[
\begin{array}{ccc}
0 & 1 & 6 \\
1 & 2 & 2 \\
2 & 0 & 1 \\
3 & 0 & 1 \\
4 & 5 & 3 \\
5 & 2 & 3
\end{array}
\]

Neighbor index:

\[
\begin{array}{ccc}
1 & 3 & 0 \\
2 & 0 & 1 \\
1 & 2 & 2 \\
5 & 2 & 3 \\
0 & 5 & 4 \\
5 & 4 & 5 \\
0 & 1 & 6 \\
5 & 3 & 7
\end{array}
\]

Edge index:

\[
\begin{array}{ccc}
1 & 6 & 5 \\
2 & 2 & 0 \\
0 & 1 & 2 \\
4 & 5 & 3 \\
2 & 3 & 4
\end{array}
\]
Generalizing Neighbor Access

```cpp
using bfs_graph = vector<vector<size_t>>;
using djk_graph = vector<vector<tuple<size_t, double>>>;

bfs_graph bfs_g;
djk_graph djk_g;

bfs(bfs_g); // This should work
```
Generalizing Neighbor Access

• For BFS, inner range stored the neighbor vertex id
  • E.g., using graph = vector<vector<size_t>>;
• For property graph, inner range stored a tuple of the neighbor vertex id and the edge property
  • E.g., using graph = vector<vector<tuple<size_t, size_t>>>;
• Use target() accessor to lift (unifying both cases)
The adjacency_list_graph concept

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

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

Type of vertex identifier (cannot be inferred)
target() is a customization-point object (CPO)
Requirements: BFS Algorithm

```cpp
template <adjacency_list Graph>
auto bfs(const Graph& graph, vertex_id_t<Graph> source);
```

- The graph G meets the requirements of the adjacency_list_graph concept.
Examples of types modeling adjacency_list_graph

```cpp
using Graph = std::vector<vector<int>>;

template <class U>
template <class U>
template <class U>
template <class U>
auto tag_invoke(const target_tag, const Graph& graph, const U& e) { 
return e;
}
```

Don’t need an actual container of containers

```cpp
using Graph = std::vector<vector<
```
Minimalist Approach: Stored Property Graph

```cpp
template <adjacency_list Graph>
auto dijkstra(const Graph& graph, vertex_id_t<Graph> source) {
    using vertex_id_type = vertex_id_t<Graph>;
    using weight_type = std::tuple_element_t<1, inner_value<Graph>>;
    using weighted_vertex = std::tuple<vertex_id_type, weight_type>;

    std::vector<weight_type> distance(size(graph), std::numeric_limits<weight_type>::max());
    distance[source] = 0;

    std::priority_queue<weighted_vertex, std::vector<weighted_vertex>,
        decltype([](auto&& a, auto&& b) { return std::get<1>(a) > std::get<1>(b); })> Q;
    Q.push({source, distance[source]});

    while (!Q.empty()) {
        auto u = std::get<0>(Q.top());
        Q.pop();

        for (auto&& e : graph[u]) {
            auto v = target(graph, e);   // neighbor vertex
            auto w = std::get<1>(e);     // edge weight
            if (distance[u] + w < distance[v]) {  // relax
                distance[v] = distance[u] + w;
                Q.push({v, distance[v]});
            }
        }
    }
    return distance;
}
```
Minimalist Approach: Index Property Graph

```cpp
template <adjacency_list Graph, typename WeightRange>
auto dijkstra(const Graph& graph, vertex_id_t<Graph> source, const WeightRange& weights) {
    using vertex_id_type = vertex_id_t<Graph>;
    using weight_type = std::ranges::range_value_t<WeightRange>;
    using weighted_vertex = std::tuple<vertex_id_type, weight_type>;

    std::vector<weight_type> distance(size(graph), std::numeric_limits<weight_type>::max());
    distance[source] = 0;

    std::priority_queue<weighted_vertex, std::vector<weighted_vertex>,
    decltype([](auto&& a, auto&& b) { return std::get<1>(a) > std::get<1>(b); })>
    Q;
    Q.push({source, distance[source]});

    while (!Q.empty()) {
        auto u = std::get<0>(Q.top()); Q.pop();

        for (auto& e : graph[u]) {
            auto v = target(e); // neighbor vertex
            auto k = std::get<1>(e); // index to edge weight
            if (distance[u] + weights[k] < distance[v]) { // relax
                distance[v] = distance[u] + weights[k];
                Q.push({v, distance[v]});
            }
        }
    }
    return distance;
}
```

Index to property is stored on the edge
Lifted Dijkstra

```cpp
template <adjacency_list_graph Graph, class WeightFunction>
autoijkstra(const Graph& graph, vertex_id_t<Graph> source, WeightFunction weights)
{
    using vertex_id_type = vertex_id_t<Graph>;
    using weight_type = std::invoke_result_t<WeightFunction, inner_value<Graph>>;
    using weighted_vertex = std::tuple<vertex_id_type, weight_type>;

    std::vector<weight_type> distance(size(graph), std::numeric_limits<weight_type>::max());
    distance[source] = 0;

    std::priority_queue<weighted_vertex, std::vector<weighted_vertex>,
        decltype([](auto&& a, auto&& b) { return std::get<1>(a) > std::get<1>(b); })>
        Q;
    Q.push({source, distance[source]});

    while (!Q.empty()) {
        auto u = std::get<0>(Q.top());
        Q.pop();

        for (auto&& e : graph[u]) {
            auto v = target(graph, e);  // neighbor vertex
            if (distance[u] + weights(e) < distance[v]) { // relax
                distance[v] = distance[u] + weights(e);
                Q.push({v, distance[v]});
            }
        }
    }
    return distance;
}
```

A weights function is used to compute weight given edge

Property is computed with whatever is stored
Example Weight Functions

```cpp
using graph = std::vector<std::list<std::tuple<size_t, double>>>;
auto e = dijkstra(G, 5UL, [](auto&& e) { return std::get<1>(e); });

using edges = std::vector<std::tuple<size_t, size_t, double>>;
using graph = std::vector<std::list<std::tuple<size_t, double>>>;
auto f = dijkstra(
    H, 5, [](auto&& e){ return std::get<2>(ospf_edges[std::get<1>(e)]); });
```
Lifted Dijkstra

Traverse inner container with neighbors and properties

Inner range has a target vertex
Requires: target()

Inner range has properties
Requires: weight()

Inner range value_type is a vertex_id

Inner range value_type is a tuple

Inner range property is stored by value as tuple element

Inner range property is stored by index as tuple element
Requirements: Dijkstra Algorithm

```cpp
template <adjacency_list_graph Graph, class WeightFunction>
auto dijkstra(const Graph& graph, vertex_id_t<Graph> source, WeightFunction weights);
```

- Graph must meet the requirements of the adjacency_list_graph concept.
- WeightFunction must meet the requirements of the invocable concept.
- weight is a function that maps from the value type of G[u] to a type that can be summed and compared.
Other Concepts

```
template <typename G>
concept degree_enumerable_graph = adjacency_list_graph<G>
    && requires (G g, vertex_id_t<G> u) {
    { degree(g[u]) } -> std::convertible_to<std::ranges::range_difference_t<G>>;
    }

template <typename G>
concept edge_list_graph = graph<G>
    && requires (G g, std::ranges::range_value_t<G> e) {
    { source(g, e) } -> std::convertible_to<vertex_id_t<G>>;
    { target(g, e) } -> std::convertible_to<vertex_id_t<G>>;
    }
```
Where Formal Lifting Still Needed

• Range adaptors
• Graph construction
• Mutable graph algorithms
• Dynamic graph algorithms
• Streaming graph algorithms

Again, lift algorithms
Range Adaptors

- Some graph “algorithms” are really traversal patterns (BFS, DFS)
- Patterns used in different ways
- Adapted in BGL with “visitors”
- Range adaptors instead present (forward) range of vertices or edges in order the algorithm traverses them
BFS is a Traversal Pattern, Not an Algorithm

```cpp
template <class Graph>
auto bfs(const Graph& graph, vertex_id_t<Graph> source) {
    using vertex_id_type = vertex_id_t<Graph>;

    std::vector<COLOR> color(size(graph));
    for (vertex_id_type u = 0; u < size(graph); ++u) {
        color[u] = WHITE;
    }
    color[source] = GREY;

    std::queue<vertex_id_type> Q;
    Q.push(source);

    while (!Q.empty()) {
        auto u = Q.front();
        Q.pop();
        for (auto&& e : graph[u]) {
            auto v = target(graph, e); // neighbor vertex
            if (color[v] == WHITE) {
                color[v] = GREY;
                Q.push(v);
            }
        }
        color[u] = BLACK;
    }
}
```

This “algorithm” doesn’t do anything

Might want to compute depth

Might want to compute predecessors (paths)

Might want to compute something else
Range Adaptors

```cpp
std::vector<std::vector<int>> costars {
    { 1, 5, 6 },
    { 7, 10, 0, 5 },
    { 4 },
    { 2, 11 },
    { 8, 9, 2 },
    { 0, 1 },
    { 7, 1 },
    { 6, 0 },
    { 4, 9 },
    { 4, 8 },
    { 7, 1 },
    { 2, 3 } );

std::vector<int> bacon_number(size(actors));
for (auto&& [u, v] : bfs_edge_range(costars, 1)) {
    bacon_number[v] = bacon_number[u] + 1;
}

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

Index Adjacency List

```
0 —> 1 —> 5 —> 6
1 —> 0 —> 6 —> 5
2 —> 4
3 —> 2 —> 11
4 —> 8 —> 9 —> 2
5 —> 0 —> 1
6 —> 7 —> 1
7 —> 6 —> 0
8 —> 4 —> 9
9 —> 4 —> 8
10 —> 7 —> 1
11 —> 2 —> 3
```

Actors

- Tom Cruise
- Kevin Bacon
- Hugo Weaving
- Cary Anne Moss
- Natalie Portman
- Jack Nicholson
- Kelly McGillis
- Harrison Ford
- Sebastian Stan
- Mila Kunis
- Michelle Pfeiffer
- Keanu Reeves
Stored Property Graph

\[ V \]
- GND
- Vdd
- n0
- n1
- n2
- out

\[ E \]
- n0  n1  C1
- Vdd GND AC
- n0  Vdd R2
- n2  Vdd R0
- GND n2  R1
- n2  Vout L1
- GND n0  C0
- n2  n1  R3

\[ E \] properties
- 1  3  C1
- 2  0  AC
- 1  2  R2
- 5  2  R0
- 0  5  R1
- 5  4  L1
- 0  1  C0
- 5  3  R3

\[ E \] edge property
- 0  → 1  C0  → 5  R1
- 1  → 2  R2  → 3  C1
- 2  → 0  AC  →
- 3  →
- 4  → 2  R0  → 4  L1  → 3  R3

Neighbor index
Index Property Graph

$V$

<table>
<thead>
<tr>
<th></th>
<th>n0</th>
<th>n1</th>
<th>C1</th>
<th>n2</th>
<th>Vdd</th>
<th>n0</th>
<th>Vdd</th>
<th>R2</th>
<th>n2</th>
<th>Vdd</th>
<th>R0</th>
<th>GND</th>
<th>n0</th>
<th>C0</th>
<th>GND</th>
<th>n0</th>
<th>R3</th>
</tr>
</thead>
<tbody>
<tr>
<td>GND</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vdd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>out</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$E$

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>0</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>7</td>
<td>5</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Edge indices

0 → 1 → 6 → 5 → 4
1 → 2 → 2 → 3 → 0
2 → 0 → 1
3 → 4 → 5 → 3 → 7

Neighbor index

Edge index
Graph Construction (Library Functions)

```cpp
template <class IndexGraph = std::vector<std::vector<size_t>>, std::ranges::random_access_range V, std::ranges::random_access_range E>
auto make_plain_graph(const V& vertices, const E& edges, bool directed = true, size_t size = 100000) {
    // Implementation...
}

template <std::ranges::random_access_range V, std::ranges::random_access_range E, std::ranges::random_access_range adjacencies>
auto make_adjacency_list_graph = std::vector<std::vector<std::tuple<size_t>>>

template <std::ranges::random_access_range V, std::ranges::random_access_range E, std::ranges::random_access_range adjacencies>
declaytype(std::tuple_cat(std::make_tuple(size_t{}), prefix), postfix) {
    // Implementation...
}

template <std::ranges::random_access_range V, std::ranges::random_access_range E, std::ranges::random_access_range adjacencies>
auto make_property_graph(const V& vertices, const E& edges, bool directed = true, size_t size = 100000) {
    // Implementation...
}

auto G = make_property_graph(ospf_vertices, ospf_edges);
```
Compressed Graph Type

- (Ala compressed sparse row matrix)
- Not a composition of containers, but has “range of ranges” interface
- Highly efficient

```cpp
template <typename vertex_type, typename edge_type, std::size_t vertex_size, std::size_t edge_size, std::size_t vertex_capacity, std::size_t edge_capacity> Attributes
class index_adjacency;
```

<table>
<thead>
<tr>
<th>V</th>
<th>E</th>
<th>Index edge list</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEA</td>
<td>MSP DTW 850</td>
<td>1 3 0</td>
</tr>
<tr>
<td>MSP</td>
<td>SLC SEA 1357</td>
<td>2 0 1</td>
</tr>
<tr>
<td>SLC</td>
<td>MSP SLC 1981</td>
<td>1 2 2</td>
</tr>
<tr>
<td>DTW</td>
<td>BOS SLC 3835</td>
<td>5 2 3</td>
</tr>
<tr>
<td>ATL</td>
<td>BOS ATL 1523</td>
<td>0 5 4</td>
</tr>
<tr>
<td>BOS</td>
<td>SEA MSP 2704</td>
<td>5 4 5</td>
</tr>
<tr>
<td></td>
<td>BOS DTW 1191</td>
<td>0 1 6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5 3 7</td>
</tr>
</tbody>
</table>

Diagram:
- Index
- Compressed
- Storage
- Representing
Six Degrees of Kevin Bacon

• The co-starring relationships between actors forms a graph
• A BFS of that graph gives each actor a “Bacon number”
• Unfortunately, the co-star graph doesn’t actually exist

• IMDB:
  • Movies table
  • Actors table
  • Movies-Actor table
Anti-Pattern

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

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

    for (auto&& [u, v] : bfs_edge_range(costars, 1)) {
        bacon_number[v] = bacon_number[u] + 1;
    }

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

    return 0;
}
```
Bipartite Graphs

Movies
A Few Good Men
Top Gun
Black Swan
V for Vendetta
The Matrix
Witness
What Lies Beneath

size = 7

Actors
Tom Cruise
Kevin Bacon
Hugo Weaving
Cary Anne Moss
Natalie Portman
Jack Nicholson
Kelly McGillis
Harrison Ford
Sebastian Stan
Mila Kunis
Michelle Pfeiffer
Keanu Reeves

size = 12

Movies - Actors
A Few Good Men - Tom Cruise
A Few Good Men - Kevin Bacon
A Few Good Men - Jack Nicholson
What Lies Beneath - Harrison Ford
What Lies Beneath - Kevin Bacon
Top Gun - Tom Cruise
Top Gun - Kelly McGillis
Witness - Harrison Ford
Witness - Kelly McGillis
Black Swan - Sebastian Stan
Black Swan - Natalie Portman
Black Swan - Mila Kunis
V for Vendetta - Hugo Weaving
V for Vendetta - Natalie Portman
The Matrix - Cary Anne Moss
The Matrix - Keanu Reeves
The Matrix - Hugo Weaving
What Lies Beneath - Michelle Pfeiffer

size = 18

Index Adjacency List <0>
0 → 0 → 1 → 5
1 → 6 → 0
2 → 8
3 → 9
4 → 11
5 → 12
6 → 10

Index Adjacency List <1>
0 → 0 → 1
1 → 6 → 0
2 → 3
3 → 4
4 → 2
5 → 1
6 → 5
7 → 6
8 → 2
9 → 6
10 → 4
11 → 4

num_vertices = (7, 12)

Index each other instead of themselves
Bipartite Graphs

Index Adjacency List <0>

Index Adjacency List <1>

Library provided

Movie-actor

Actor-movie

Actor-actor
Six Degrees of Kevin Bacon (IMDB version)

```cpp
#include "imdb-graph.hpp"

auto & [G, H] = make_plain_bipartite_graphs<movies, actors, movies_actors>;

auto L = join(G, H);
auto M = join(H, G);

size_t kevin_bacon = 1;
std::vector<size_t> distance(L.size());
std::vector<size_t> parents(L.size());
std::vector<size_t> together_in(L.size());

for (auto & [u, v, k] : bfs_edge_range(L, kevin_bacon)) {
  distance[v] = distance[u] + 1;
  parents[v] = u;
  together_in[v] = k;
}
```

Tables (vectors of strings)
Six Degrees of Kevin Bacon

    // Iterate through all actors (other than Kevin Bacon)
    for (size_t i = 0; i < actors.size(); ++i) {
        if (i != kevin_bacon) {
            auto bacon_number = distance[i];
            std::cout << actors[i] << " has a bacon number of " << distance[i] << std::endl;

            auto k = i;
            size_t d = distance[k];
            while (k != kevin_bacon) {
                std::cout << " " << actors[k] << " starred with " << actors[parents[k]] << " in "
                    << movies[together_in[k]] << std::endl;
                k = parents[k];
                if (d-- == 0) {
                    break;
                }
            }
            std::cout << std::endl;
        }
    }

## For Those Interested to Know

<table>
<thead>
<tr>
<th>Name</th>
<th>Bacon Number</th>
<th>Connections</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kevin Bacon</td>
<td>0</td>
<td>Tom Cruise starred with Kevin Bacon in <em>What Lies Beneath</em></td>
</tr>
<tr>
<td>Tom Cruise</td>
<td>1</td>
<td>Harrison Ford starred with Kevin Bacon in <em>What Lies Beneath</em></td>
</tr>
<tr>
<td>Hugo Weaving</td>
<td>3</td>
<td>Sebastian Stan starred with Natalie Portman in <em>Black Swan</em></td>
</tr>
<tr>
<td>Carrie-Anne Moss</td>
<td>4</td>
<td>Mila Kunis starred with Natalie Portman in <em>Black Swan</em></td>
</tr>
<tr>
<td>Natalie Portman</td>
<td>2</td>
<td>Michelle Pfeiffer starred with Kevin Bacon in <em>What Lies Beneath</em></td>
</tr>
<tr>
<td>Jack Nicholson</td>
<td>1</td>
<td>Keanu Reeves starred with Hugo Weaving in <em>The Matrix</em></td>
</tr>
<tr>
<td>Kelly McGillis</td>
<td>2</td>
<td>Julia Roberts starred with Kevin Bacon in <em>What Lies Beneath</em></td>
</tr>
</tbody>
</table>

- Hugo Weaving starred with Natalie Portman in *V for Vendetta*
- Natalie Portman starred with Julia Roberts in *Closer*
- Julia Roberts starred with Kevin Bacon in *Flatliners*

- Jack Nicholson starred with Kevin Bacon in *A Few Good Men*
- Kelly McGillis starred with Tom Cruise in *Top Gun*
- Tom Cruise starred with Kevin Bacon in *A Few Good Men*
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>BGL Input Requirement on Graph Concept</th>
<th>Other Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>BC clustering</td>
<td>Vertex List Graph and Incidence Graph (and Edge List Graph and Mutable Graph)</td>
<td></td>
</tr>
<tr>
<td>bellman ford</td>
<td>Edge List Graph (and Vertex List Graph)</td>
<td>directed or undirected</td>
</tr>
<tr>
<td>bfs</td>
<td>Vertex List Graph and Incidence Graph</td>
<td>directed or undirected</td>
</tr>
<tr>
<td>bicc</td>
<td>Vertex List Graph and Incidence Graph</td>
<td>undirected</td>
</tr>
<tr>
<td>BK Max Flow</td>
<td>Vertex List Graph and Incidence Graph and Edge List Graph</td>
<td>directed</td>
</tr>
<tr>
<td>Brandes BC</td>
<td>Vertex List Graph and Incidence Graph</td>
<td></td>
</tr>
<tr>
<td>CC</td>
<td>Vertex List Graph and Incidence Graph</td>
<td>undirected</td>
</tr>
<tr>
<td>cuthill mckee</td>
<td>Incidence Graph</td>
<td>directed or undirected</td>
</tr>
<tr>
<td>delta stepping</td>
<td>Vertex List Graph and Incidence Graph</td>
<td></td>
</tr>
<tr>
<td>dfs</td>
<td>Vertex List Graph and Incidence Graph</td>
<td></td>
</tr>
<tr>
<td>dijkstra</td>
<td>Vertex List Graph and Incidence Graph</td>
<td>weighted</td>
</tr>
<tr>
<td>dominator tree</td>
<td>Vertex List Graph and Bidirectional Graph</td>
<td>directed (?)</td>
</tr>
<tr>
<td>EK Max Flow</td>
<td>Vertex List Graph and Incidence Graph</td>
<td>directed</td>
</tr>
<tr>
<td>floyd warshall</td>
<td>Vertex List Graph (and Vertex And Edge List Graph)</td>
<td>directed or undirected</td>
</tr>
<tr>
<td>johnson all pairs</td>
<td>Vertex List Graph and Incidence graph and Edge List Graph</td>
<td>directed or undirected</td>
</tr>
<tr>
<td>Kruskal</td>
<td>Vertex List Graph and Edge List Graph</td>
<td>undirected</td>
</tr>
<tr>
<td>max card matching</td>
<td>Vertex And Edge List Graph and Incidence Graph</td>
<td>undirected</td>
</tr>
<tr>
<td>max weighted matching</td>
<td>Vertex And Edge List Graph and Incidence Graph</td>
<td>undirected</td>
</tr>
<tr>
<td>min degree ordering</td>
<td>Vertex List Graph and Incidence Graph and Adjacency Graph and Mutable Graph</td>
<td>directed</td>
</tr>
<tr>
<td>page rank</td>
<td>Vertex List Graph and Incidence Graph</td>
<td>directed</td>
</tr>
<tr>
<td>PR Max Flow</td>
<td>Vertex List Graph</td>
<td>directed</td>
</tr>
<tr>
<td>Prim</td>
<td>Vertex List Graph and Incidence Graph</td>
<td>undirected</td>
</tr>
<tr>
<td>RCM</td>
<td>Incidence Graph</td>
<td>undirected</td>
</tr>
<tr>
<td>SCC</td>
<td>Vertex List Graph and Incidence Graph</td>
<td>directed</td>
</tr>
<tr>
<td>stoer wagner min cut</td>
<td>Vertex List Graph and Incidence Graph</td>
<td>undirected</td>
</tr>
<tr>
<td>topological sort</td>
<td>Vertex List Graph and Incidence Graph</td>
<td>dag</td>
</tr>
<tr>
<td>transitive closure</td>
<td>Vertex List Graph and Adjacency Graph and Adjacency Matrix</td>
<td>directed</td>
</tr>
<tr>
<td>triangle counting</td>
<td>Vertex List Graph and Incidence Graph</td>
<td>undirected</td>
</tr>
</tbody>
</table>
Boost Graph Library

- The BGL concepts were derived with same generic programming process
- Our concepts cover the same space
- But has streamlined concepts (rather than Swiss Army Knife)

<table>
<thead>
<tr>
<th>BGL Concept</th>
<th>Purpose</th>
<th>Functions</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adjacency Graph</td>
<td>Iterate neighbors of vertex</td>
<td>adjacent_vertices(v, g)</td>
<td></td>
</tr>
<tr>
<td>Adjacency Matrix</td>
<td>Random access to edge</td>
<td>edge(u, v, g)</td>
<td></td>
</tr>
<tr>
<td>BiDirectional Graph</td>
<td>Iterate over in edges</td>
<td>in_edges(u, g) in_degree(u, g)</td>
<td></td>
</tr>
<tr>
<td>Edge List Graph</td>
<td>Iterate all edges in graph</td>
<td>edges(g), num_edges(g)</td>
<td>source(e, g), target(e, g)</td>
</tr>
<tr>
<td>Incidence Graph</td>
<td>Access neighbors of vertex, including edges</td>
<td>out_edges(u, g) out_degree(u, g)</td>
<td>source(e, g), target(e, g)</td>
</tr>
<tr>
<td>Vertex And Edge List Graph</td>
<td>Refines Vertex List Graph and Edge List Graph</td>
<td>out_edges(u, g) out_degree(u, g)</td>
<td>source(e, g), target(e, g)</td>
</tr>
<tr>
<td>Vertex List Graph</td>
<td>Iterate all vertices in graph</td>
<td>vertices(g), num_vertices(g)</td>
<td></td>
</tr>
</tbody>
</table>
Proposal for Standard Graph Library: P1907r3

• P1709r3d: Graph Library
  • Phil Ratzloff (primary contact), et al
• Currently in SG19 working group
• Hope to standardize for C++23
• If you are interested, participate!
Graph BLAS

• Generalized linear algebra operations based on correspondence between graphs and sparse matrices
• Generalized sparse matrix by vector product
• Generalized sparse matrix by sparse matrix product
• Element-wise operations
• Masking
Sparse Matrix-Matrix Product

```cpp
template class SparseMatrix1, class SparseMatrix2,
class SparseMatrix3, class SparseMatrix4,
class UnaryFunction1, class UnaryFunction2,
class BinaryFunction1, class BinaryFunction2>
void mxm (const SparseMatrix1 &A, const SparseMatrix2 &B,
          const SparseMatrix3 &M, SparseMatrix4 &C,
          UnaryFunction1 initialize, UnaryFunction2 merge,
          BinaryFunction1 combine, BinaryFunction2 reduce);
```

• Implementation, requirements TBD.

• (cf. “GraphBLAS: Building a C++ Matrix API for Graph Algorithms”)
Sparse Matrix-Matrix Product

```cpp
<template class SparseMatrix1, class Vector1,
    class Vector2, class Vector3,
    class UnaryFunction1, class UnaryFunction2,
    class BinaryFunction1, class BinaryFunction2>

void mxv (const SparseMatrix1 &A, const Vector1 &B,
           const Vector2 &M, Vector3 &C,
           UnaryFunction1 initialize, UnaryFunction2 merge,
           BinaryFunction1 combine, BinaryFunction2 reduce);
```
Abstraction Penalty Measurements

• Compare different traversals of graph (sparse matrix-vector product)

```c++
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];
    }
}
```

```c++
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;
}
```

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

• Continued...

```cpp
auto per = make_edge_range<0>(graph);
for (auto&& j : per) {
    y[std::get<0>(j)] += x[std::get<1>(j)] * std::get<2>(j);
}
```

```cpp
auto per = make_edge_range<0>(graph);
std::for_each(per.begin(), per.end(), [&] (auto&& j) {
    y[std::get<0>(j)] += x[std::get<1>(j)] * std::get<2>(j);
});
```
Abstraction penalty

![Graph showing runtime in ms for different operations and datasets]


- **Runtime in ms**

- **raw**, **iterator**, **for_each**, **neighbor_range**, **range**
Abstraction Penalty

![Bar chart showing runtime in ms for different data structures and abstractions. The x-axis represents different datasets: circuit5M-i9, circuit5M-M1, GAP-road-i9, GAP-road-M1, hugebubbles-i9, hugebubbles-M1. The y-axis represents runtime in ms. The chart compares the following data structures:

- struct_of_array
- vector_of_list
- vector_of_vector
- vector_of_forward_list]
Performance Comparisons

Normalized Time (relative to NWgr)

- BFS
- GAPBS
- GraphIt
- Galois
- NWgr

Graphs for different datasets:
- Web
- Twitter
- Road
- Kron
- Urand
Performance Comparisons

Normalized Time (relative to NWgr)

- GAPBS
- GraphIt
- Galois
- NWgr

SSSP

Web
Twitter
Road
Kron
Urando
Lessons Learned

- The standard library is sufficient for (sequential) graph algorithms
- **A graph is a random-access range of forward ranges**
- (More capabilities are needed to support parallel graph algorithms, e.g., concurrent containers, more control over parallel partitioning)
- Ranges + concepts = synergy
To Find Out More

• Gather Town directly after this talk
• https://github.com/lums658/cppcon21
  • All code from these slides (and more)
• P1907R3d
• Proposed std::graph
• NWGraph pre-print (upon request)
• NWGraph (release imminent)
• andrew.lumsdaine@tiledb.com
• phil.ratzloff@sas.com
Creative Commons BY-NC-SA 4.0 License

© Andrew Lumsdaine and Phil Ratzloff, 2021

Except where otherwise noted, this work is licensed under

https://creativecommons.org/licenses/by-nc-sa/4.0/