Generic Graph Libraries in C++20

ANDREW LUMSDAINE, PHIL RATZLOFF

Cppcon
The C++ Conference
2021
October 24-29

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.

[tile]DB

- 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

SAS Institute

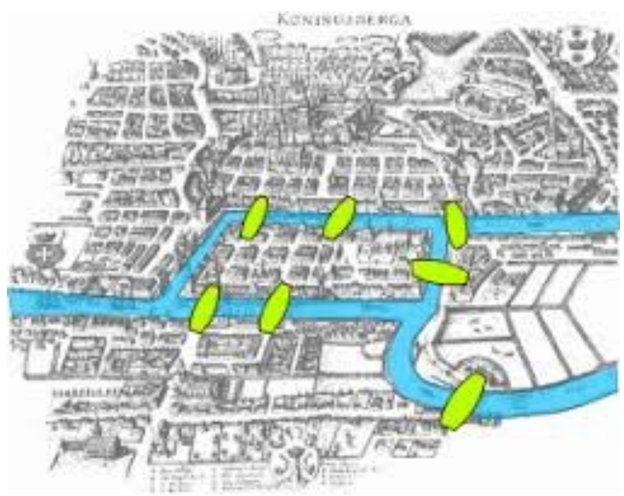
Acknowledgments and Disclaimers

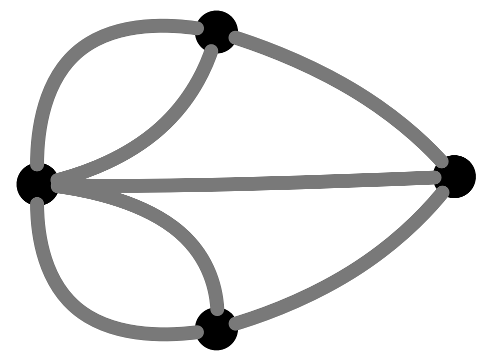
- Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation, TileDB, Inc., SAS Institute, Pacific Northwest National Laboratory, U.S. Department of Energy, the University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

National Science Foundation

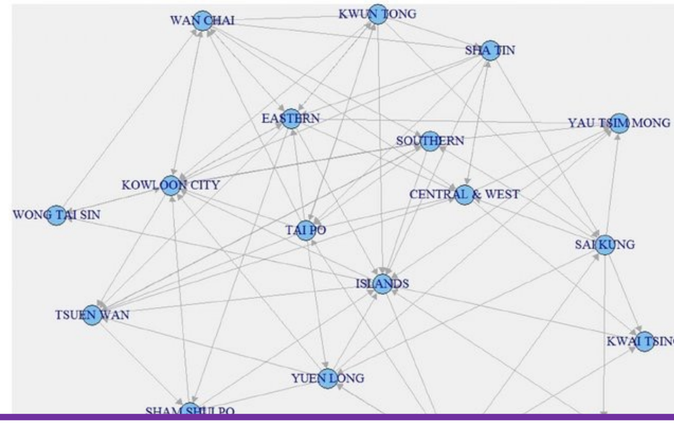
Graphs Are Fundamental Abstractions

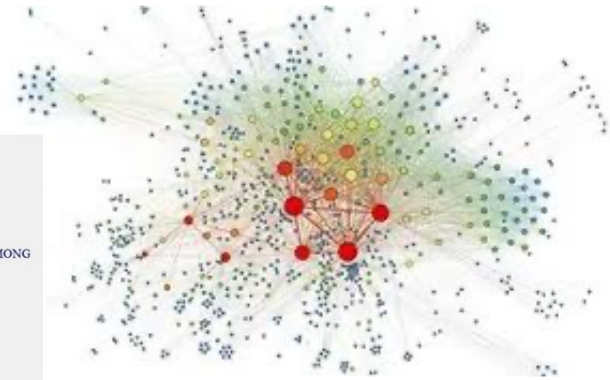
- Graphs model relationships between elements of a data set
- Without regard to what the data set actually is
- Graph theoretical (abstract) results can be applied to many different practical (concrete) problems – theory reuse
- Goes hand-in-glove with goals of generic software libraries

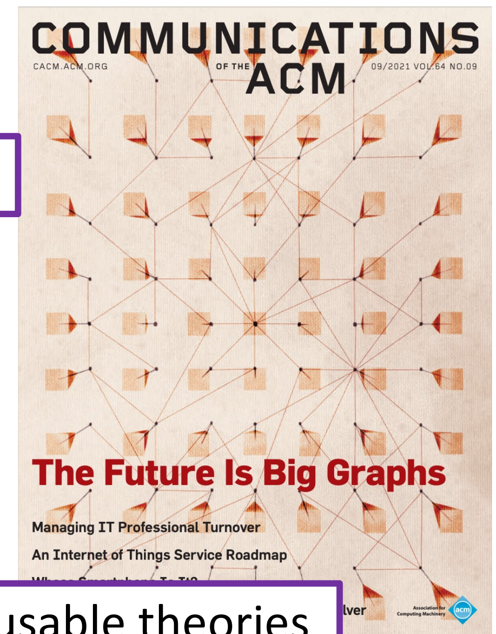




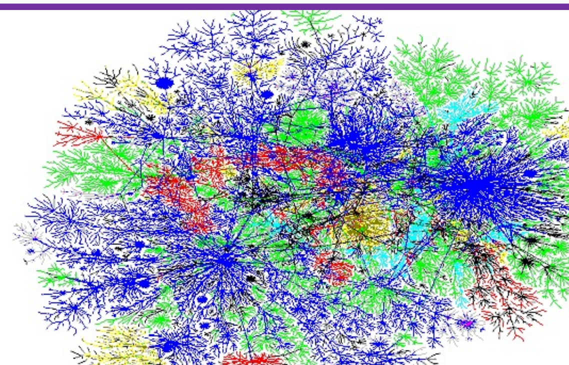
Graphs Are Ubiquitous







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

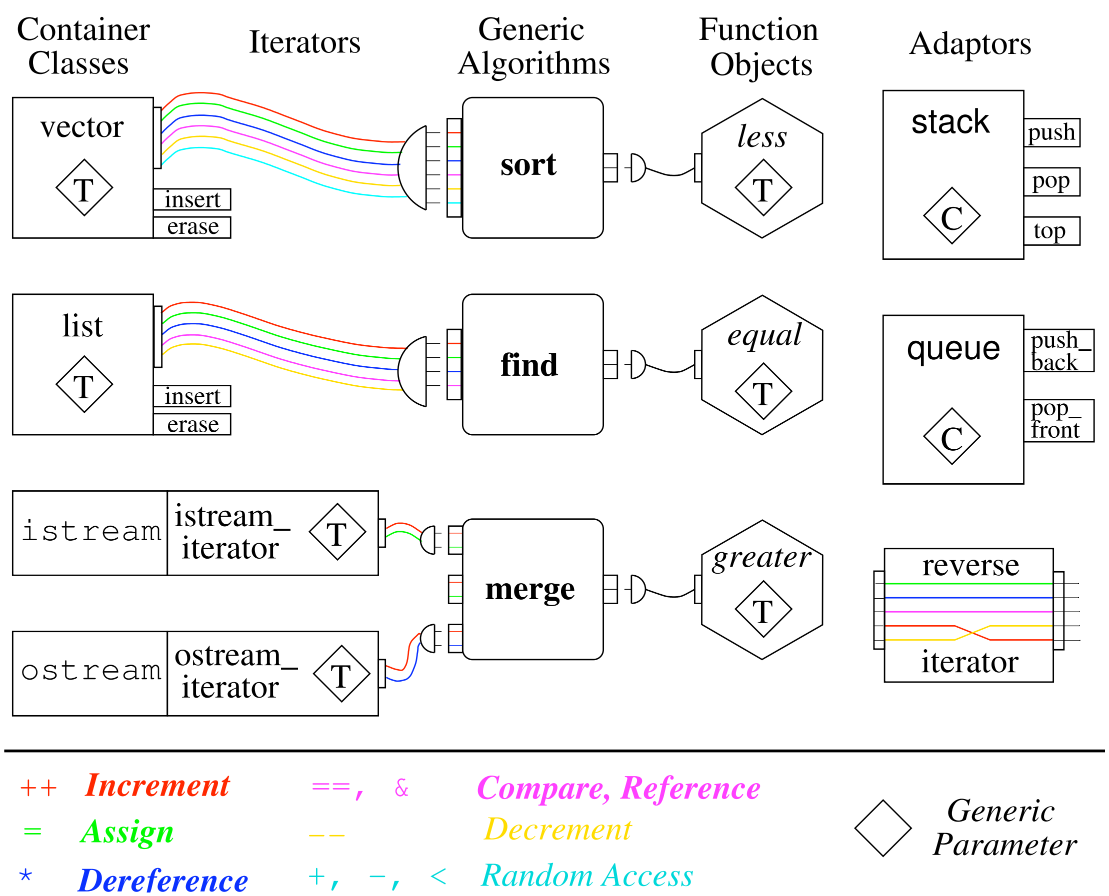
- Generic programming is a sub-discipline of computer science that deals with finding abstract representations of efficient algorithms, data structures, and other software concepts, and with their systematic organization.
- The goal of generic programming is to express algorithms and data structures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

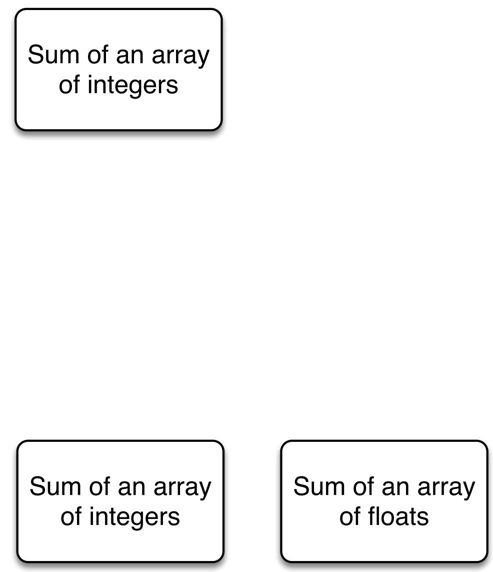
STL Architecture



Lifting Summation

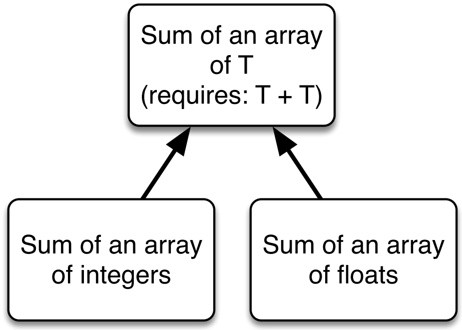
```
int sum(int* array, int n) {
  int s = 0;
  for (int i = 0; i < n; ++i)
    s = s + array[i];
  return s;
}
```

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

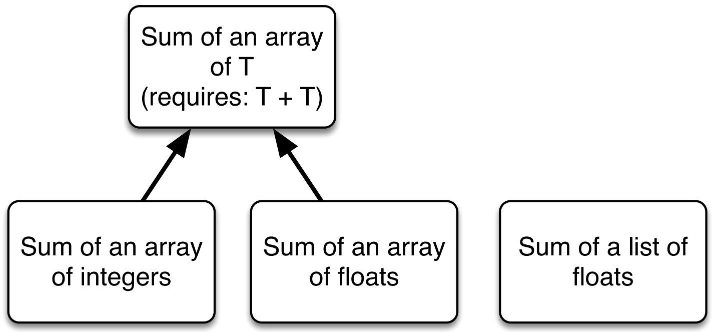


Lifting Summation

```
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;
}
```

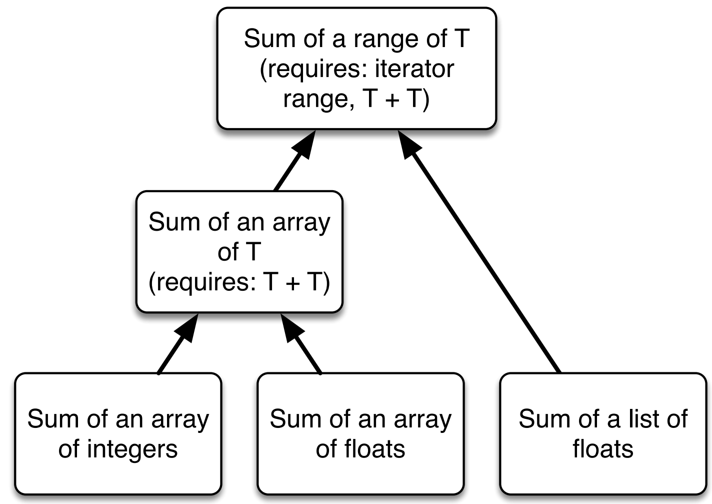



```
double sum(node* first, node* last) {
  double s = 0;
  while (first != last) {
    s = s + first->data;
    first = first->next;
  }
  return s;
}
```



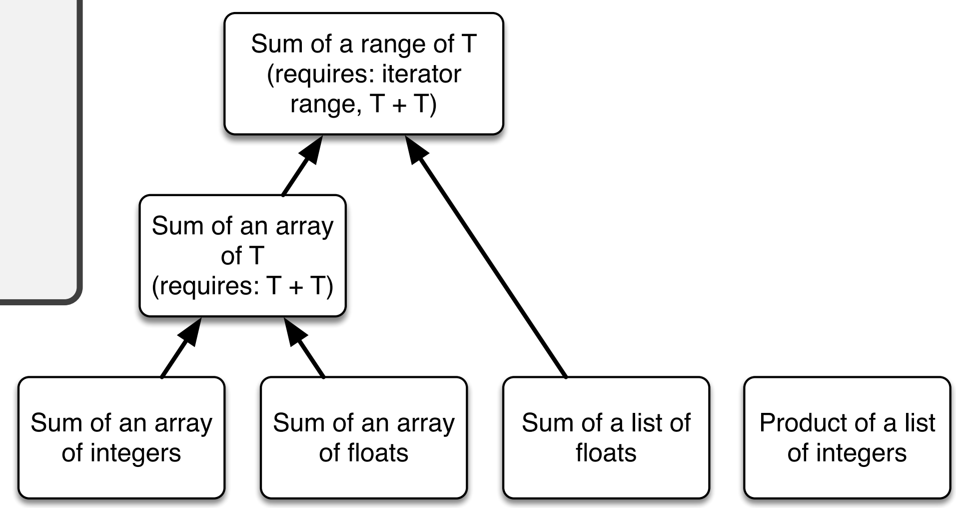
Lifting Summation

```
template <class InputIterator>
value_type sum(InputIterator first,
               InputIterator last) {
  value_type s = 0;
  while (first != last)
    s = s + *first++;
  return s;
}
```



Lifting Summation

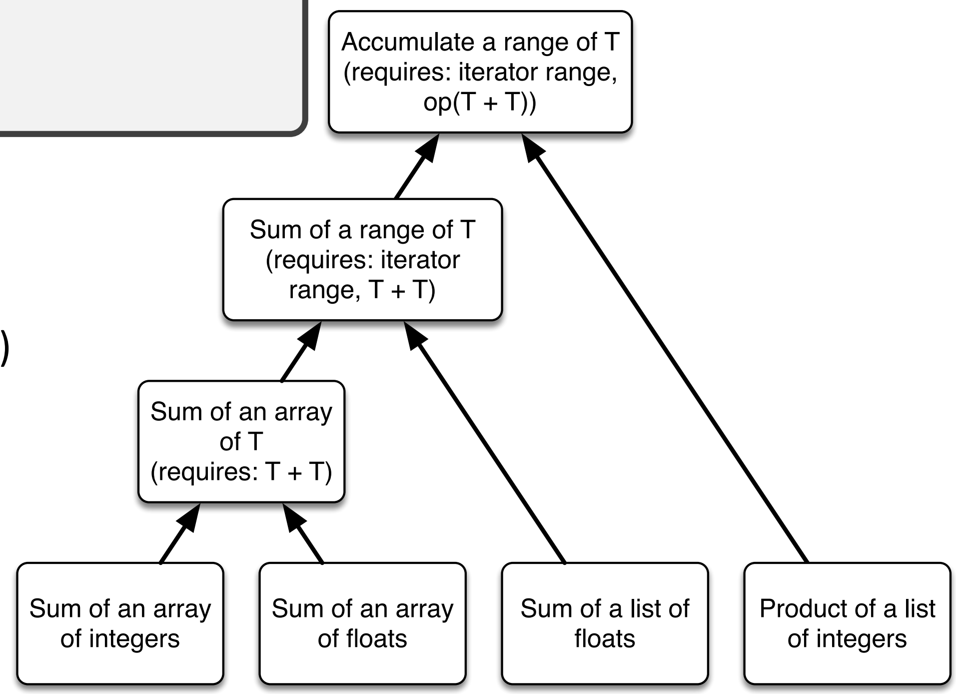
```
float product(node* first, node* last) {
  float s = 1;
  while (first != last) {
    s = s * first->data;
    first = first->next; }
  return s;
}
```



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

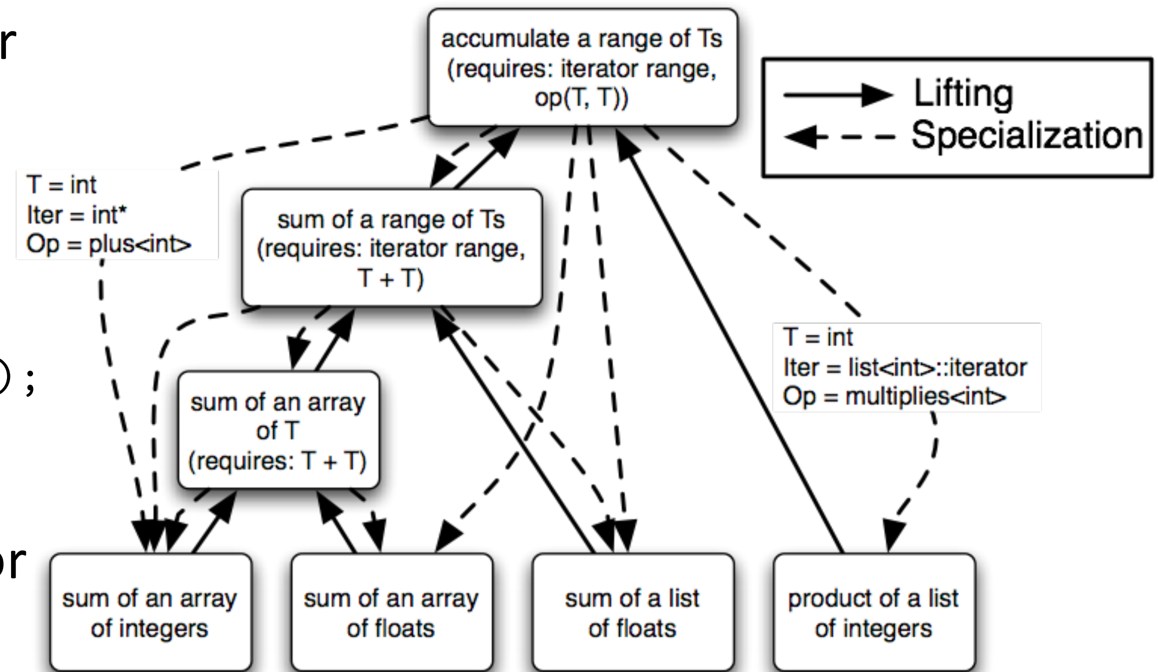


Lifting and Specialization

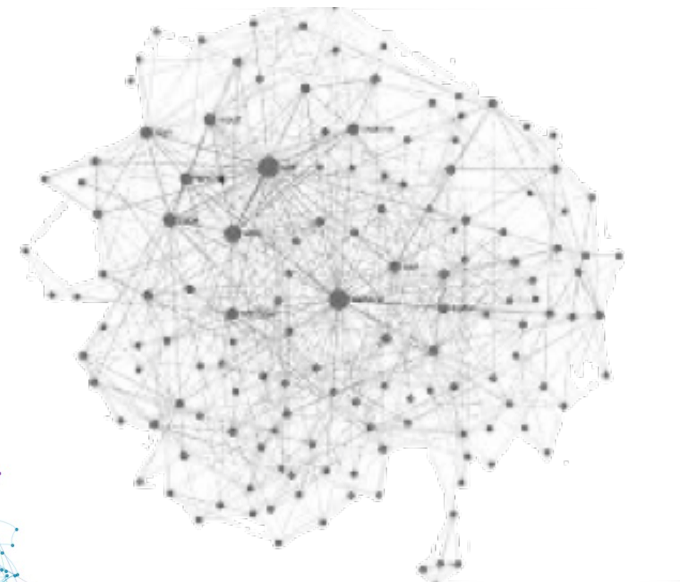
- Specialization is **dual** to lifting
- Synthesizes efficient code for a particular use of a generic algorithm:

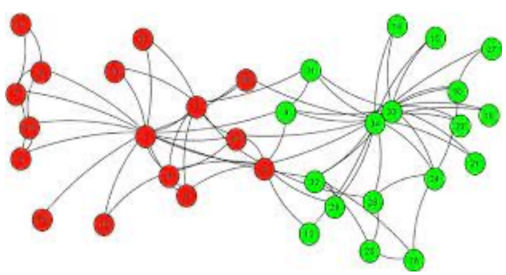
```
int array[20];
accumulate(array, array + 20,
           0, std::plus<int>());
```

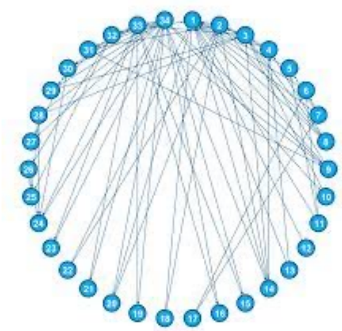
- … generates the ***same code*** as our initial sum function for integer arrays.



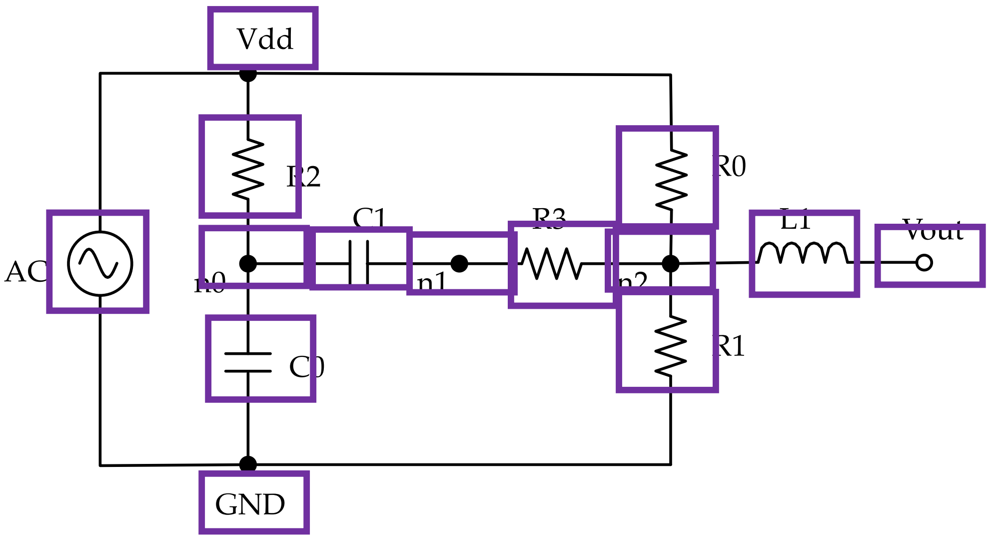
Let’s Apply the Generic Programming Process to Graphs







Graphs, Data, Properties, and Property Graphs

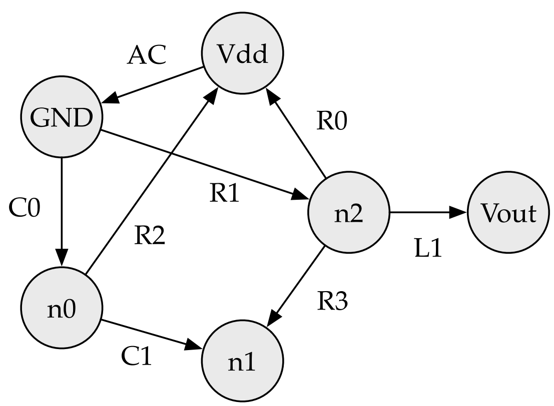


V

GND
Vdd
n0
n1
n2
out

E

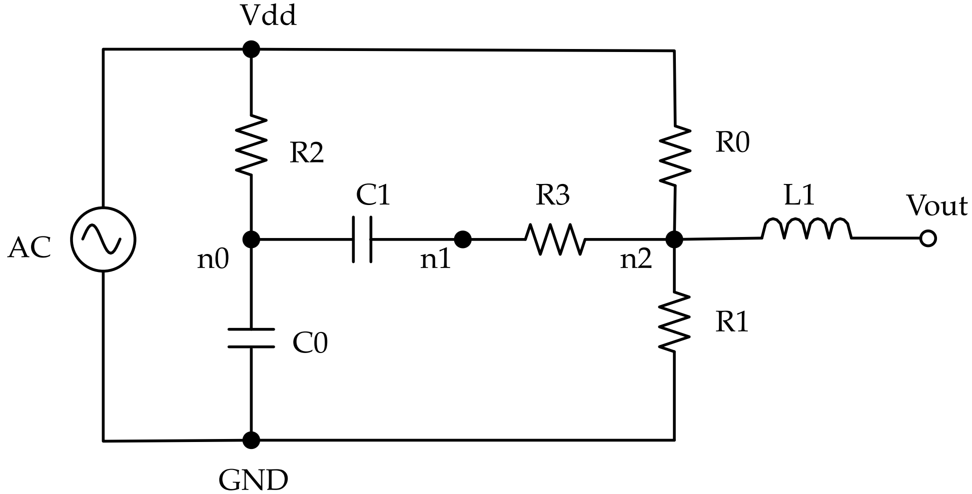
n0	n1	C1
Vdd	GND	AC
n0	Vdd	R2
n2	Vdd	R0
GND	n2	R1
n2	Vout	L1
GND	n0	C0
n2	n1	R3



$$
\begin{aligned}
G &= \{V, E\} \\
V &= \{\mathrm{GND}, \mathrm{Vdd}, \mathrm{n0}, \mathrm{n1}, \mathrm{n2}, \mathrm{Vout}\} \\
E &= \{(\mathrm{n0}, \mathrm{n1}), (\mathrm{Vdd}, \mathrm{GND}), \\
&\quad (\mathrm{n0}, \mathrm{Vdd}), (\mathrm{n2}, \mathrm{Vdd}), \\
&\quad (\mathrm{GND}, \mathrm{n2}), (\mathrm{n2}, \mathrm{Vout}), \\
&\quad (\mathrm{GND}, \mathrm{n0}), (\mathrm{n2}, \mathrm{n1})\}
\end{aligned}
$$

This is what is important

Graphs, Data, Properties, and Property Graphs

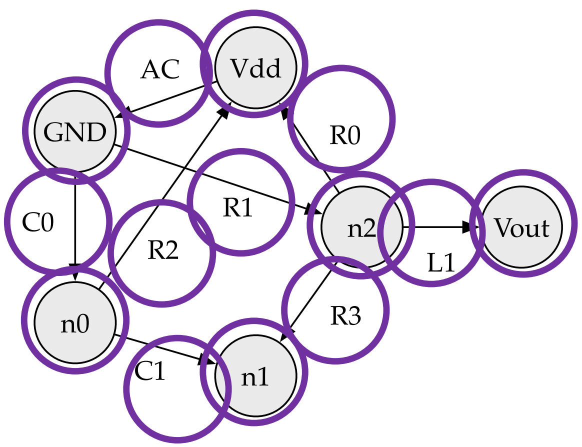


V

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



$$G = \{V, E\}$$

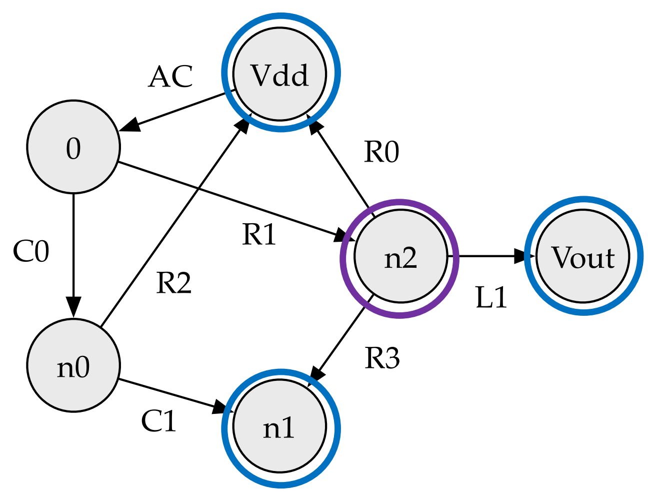
$$V = \{\mathrm{GND}, \mathrm{Vdd}, \mathrm{n0}, \mathrm{n1}, \mathrm{n2}, \mathrm{Vout}\}$$

$$E = \{(\mathrm{n0}, \mathrm{n1}), (\mathrm{Vdd}, \mathrm{GND}), (\mathrm{n0}, \mathrm{Vdd}), (\mathrm{n2}, \mathrm{Vdd}), (\mathrm{GND}, \mathrm{n2}), (\mathrm{n2}, \mathrm{Vout}), (\mathrm{GND}, \mathrm{n0}), (\mathrm{n2}, \mathrm{n1})\}$$

This is what is important

Traversal

- Traversal 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)

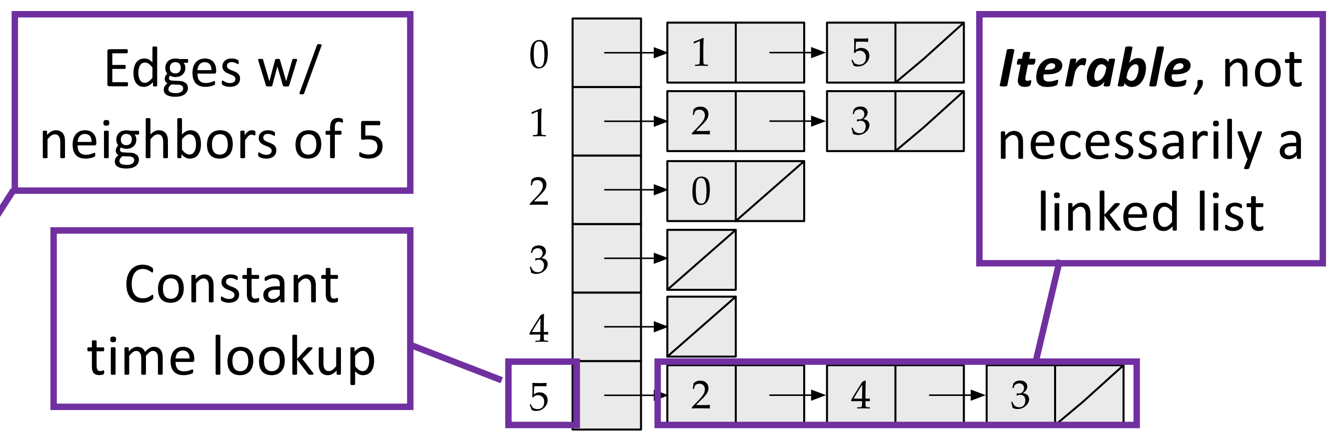


$$
\begin{aligned}
G &= \{V, E\} \\
V &= \{0, \text{Vdd}, \text{n0}, \text{n1}, \text{n2}, \text{Vout}\} \\
E &= \{(\text{n0}, \text{n1}), (\text{Vdd}, 0), \\
&\quad (\text{n0}, \text{Vdd}), (\text{n2}, \text{Vdd}), \\
&\quad (0, \text{n2}), (\text{n2}, \text{Vout}), \\
&\quad (0, \text{n0}), (\text{n2}, \text{n1})\}
\end{aligned}
$$

Adjacency "List"

- An *adjacency list* $Adj(G)$ is an array of size $|V|$
- Each entry $Adj(G)[u]$ 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 $v \in Adj(G)[u]$ **and** $u \in Adj(G)[v]$

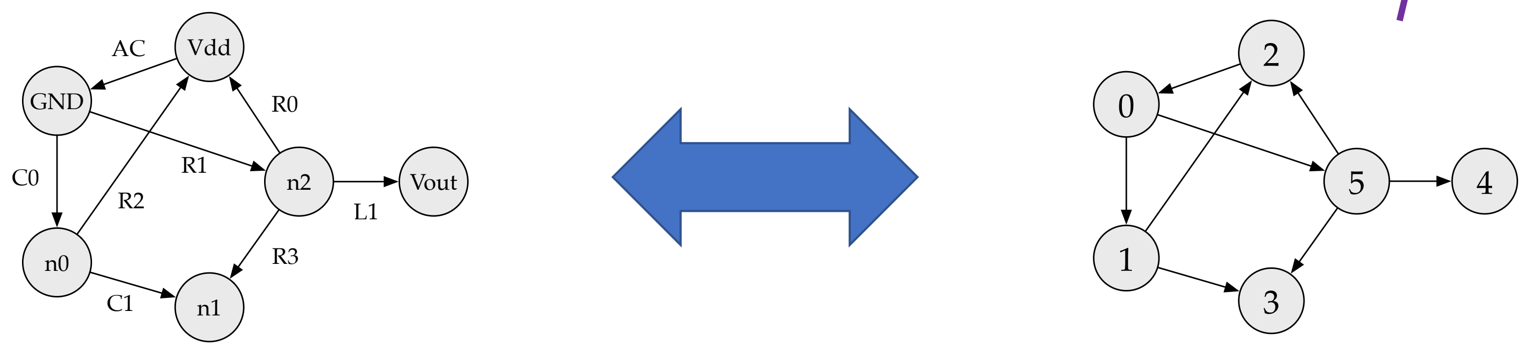
$$\begin{aligned} G &= \{V, E\} \\ V &= \{0, 1, 2, 3, 4, 5\} \\ E &= \{(1,3), (2,0), \\ &\quad (1,2), (5,2), \\ &\quad (0,5), (5,4), \\ &\quad (0,1), (5,3)\} \end{aligned}$$



Index Graphs

- Did you notice the sleight of hand?

Structure, not data



$$
\begin{aligned}
G &= \{V, E\} \\
V &= \{0, \mathrm{Vdd}, \mathrm{n0}, \mathrm{n1}, \mathrm{n2}, \mathrm{Vout}\} \\
E &= \{(\mathrm{n0}, \mathrm{n1}), (\mathrm{Vdd}, 0), \\
&\quad (\mathrm{n0}, \mathrm{Vdd}), (\mathrm{n2}, \mathrm{Vdd}), \\
&\quad (0, \mathrm{n2}), (\mathrm{n2}, \mathrm{Vout}), \\
&\quad (0, \mathrm{n0}), (\mathrm{n2}, \mathrm{n1})\}
\end{aligned}
$$

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

Principle: Graphs Represent Stru

Algorithms use this

Structure, not data

V

GND
Vdd
n0
n1
n2
out

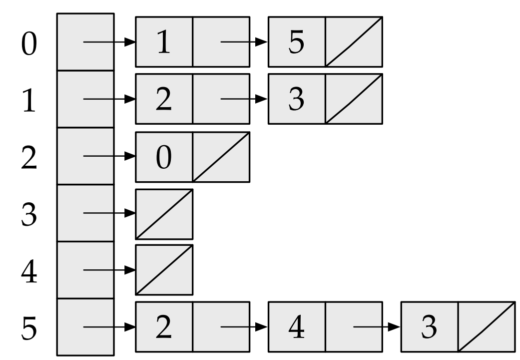
E

n0	n1	C1
Vdd	GND	AC
n0	Vdd	R2
n2	Vdd	R0
GND	n2	R1
n2	Vout	L1
GND	n0	C0
n2	n1	R3

Index
Edge List

1	3
2	0
1	2
5	2
0	5
5	4
0	1
5	3

Index Adjacency List



Structure is in here (implicitly)

Library provided

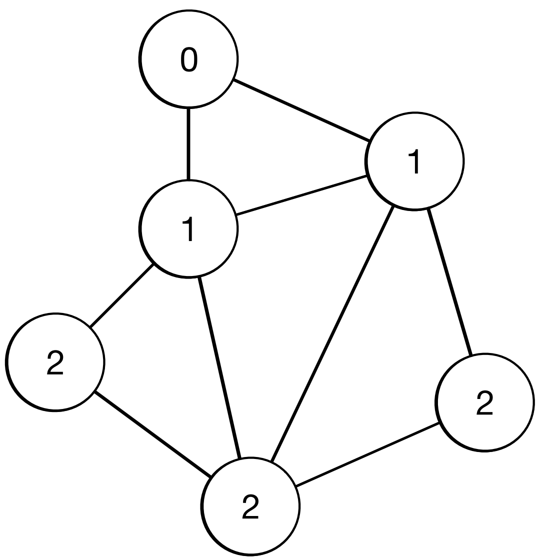
Library provided

User shouldn't be building this manually

cf: neo4j

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)
for each vertex u ∈ V(G)
    color[u] ← WHITE
color[s] ← GRAY
Q ← ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u ← DEQUEUE(Q)
    for each v ∈ Adj(G)[u]
        if color[v] = WHITE
            color[v] ← GRAY
            ENQUEUE(Q, v)
    color[u] ← 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

```
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;
  }
}
```

Enumerate vertices

Vertices can index into random access containers

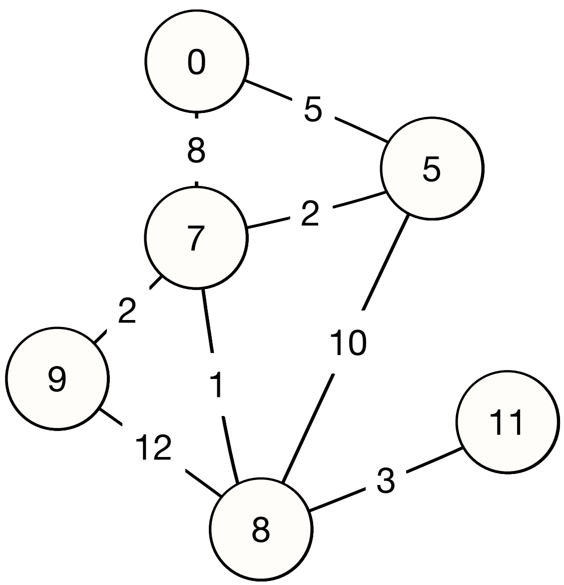
Enumerate neighbor vertices

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 `size(G)-1`, inclusive.

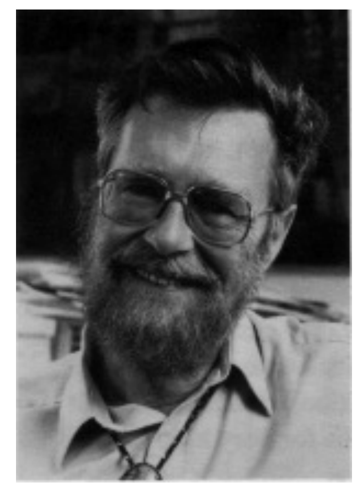
Adjacency-List Algorithms II: Dijkstra's Algorithm

- 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



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

```
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

```
DIJKSTRA(G, w, s)
for each vertex u ∈ V(G)
    d[u] ← ∞
    π[u] ← NIL
d[s] ← 0
Q ← V(G)
while Q ≠ ∅
    u ← EXTRACT-MIN(Q)
    for each v ∈ Adj(G)[u]
        if d[v] > d[u] + w(u, v)
            d[v] ← d[u] + w(u, v)
            π[v] = u
```

Enumerate vertices

Vertices can random access into containers

Enumerate neighbor vertices

Compute properties from vertex pairs (aka “edge”)

Stored Property Graph

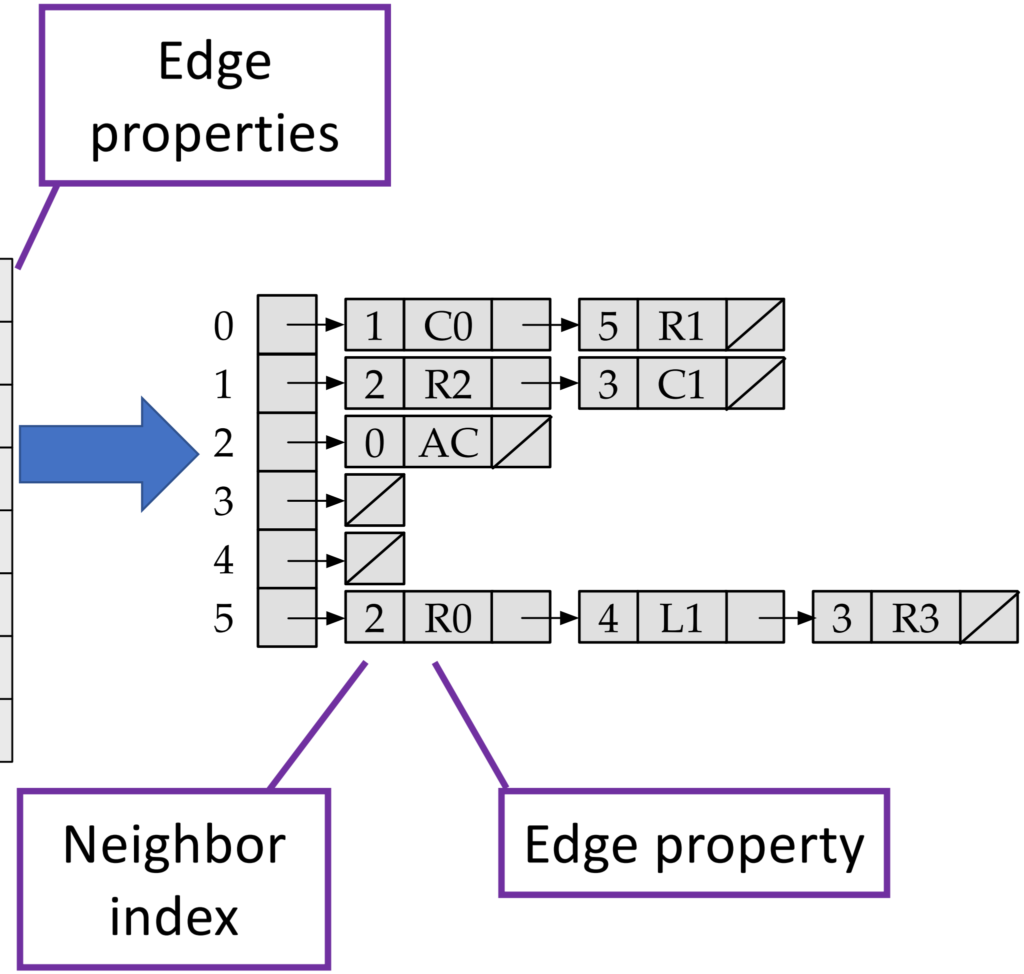
V

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

1	3	C1
2	0	AC
1	2	R2
5	2	R0
0	5	R1
5	4	L1
0	1	C0
5	3	R3



Index Property Graph

V

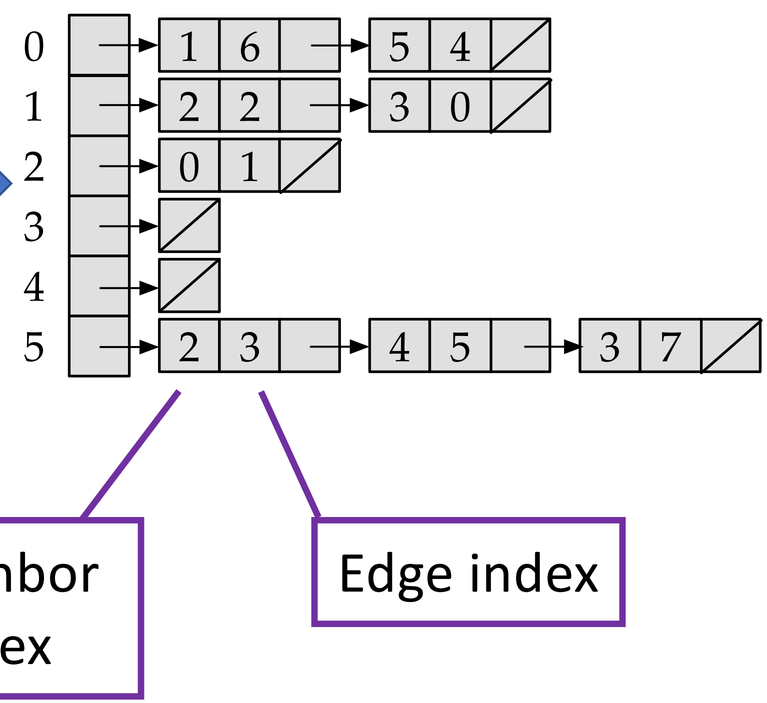
GND
Vdd
n0
n1
n2
out

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

Edge indices

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



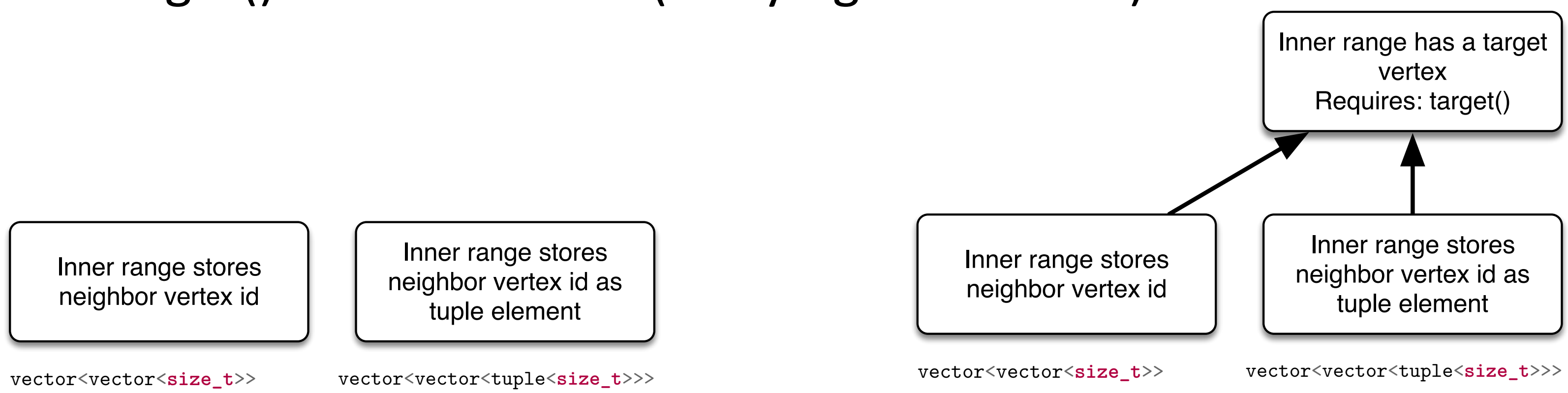
Generalizing Neighbor Access

```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);

bfs(djk_g);  // This should work
```

```
g++-11 -std=c++20 bfs_eg.cpp
In file included from bfs_eg.cpp:5:
bfs.hpp: In instantiation of 'auto bfs(const Graph&, vertex_id_t<Graph>) [with Graph = std::vector<std::__cxx11::list<std::tuple<long unsigned int, double>,
std::allocator<std::tuple<long unsigned int, double> > >, std::allocator<std::__cxx11::list<std::tuple<long unsigned int, double>, std::allocator<std::tuple<long unsigned int,
double> > > > >; vertex_id_t<Graph> = long unsigned int]':
bfs_eg.cpp:17:6:   required from here
bfs.hpp:36:16: error: no match for 'operator[]' (operand types are 'std::vector<COLOR>' and 'const std::tuple<long unsigned int, double>')
   36 |       if (color[v] == WHITE) {
      |           ~~~~~^
In file included from /opt/homebrew/Cellar/gcc/11.2.0/include/c++/11.1.0/vector:67,
                 from stdgraphs.hpp:9,
                 from bfs_eg.cpp:3:
/opt/homebrew/Cellar/gcc/11.2.0/include/c++/11.1.0/bits/stl_vector.h:1043:7: note: candidate: 'std::vector<_Tp, _Alloc>::reference std::vector<_Tp,
_Alloc>::operator[](std::vector<_Tp, _Alloc>::size_type) [with _Tp = COLOR; _Alloc = std::allocator<COLOR>; std::vector<_Tp, _Alloc>::reference = COLOR&; std::vector<_Tp,
_Alloc>::size_type = long unsigned int]'
 1043 |       operator[](size_type __n) _GLIBCXX_NOEXCEPT
      |       ^~~~~~~~
/opt/homebrew/Cellar/gcc/11.2.0/include/c++/11.1.0/bits/stl_vector.h:1043:28: note:   no known conversion for argument 1 from 'const std::tuple<long unsigned int, double>' to
'std::vector<COLOR>::size_type' {aka 'long unsigned int'}
 1043 |       operator[](size_type __n) _GLIBCXX_NOEXCEPT
      |                  ~~~~~~~~~~^~~
/opt/homebrew/Cellar/gcc/11.2.0/include/c++/11.1.0/bits/stl_vector.h:1061:7: note: candidate: 'std::vector<_Tp, _Alloc>::const_reference std::vector<_Tp,
_Alloc>::operator[](std::vector<_Tp, _Alloc>::size_type) const [with _Tp = COLOR; _Alloc = std::allocator<COLOR>; std::vector<_Tp, _Alloc>::const_reference = const COLOR&;
std::vector<_Tp, _Alloc>::size_type = long unsigned int]'
 1061 |       operator[](size_type __n) const _GLIBCXX_NOEXCEPT
      |       ^~~~~~~~
/opt/homebrew/Cellar/gcc/11.2.0/include/c++/11.1.0/bits/stl_vector.h:1061:28: note:   no known conversion for argument 1 from 'const std::tuple<long unsigned int, double>' to
'std::vector<COLOR>::size_type' {aka 'long unsigned int'}
 1061 |       operator[](size_type __n) const _GLIBCXX_NOEXCEPT
      |                  ~~~~~~~~~~^~~
In file included from bfs_eg.cpp:5:
bfs.hpp:37:14: error: no match for 'operator[]' (operand types are 'std::vector<COLOR>' and 'const std::tuple<long unsigned int, double>')
   37 |         color[v] == GREY;
      |         ~~~~~^
In file included from /opt/homebrew/Cellar/gcc/11.2.0/include/c++/11.1.0/vector:67,
                 from stdgraphs.hpp:9,
                 from bfs_eg.cpp:3:
/opt/homebrew/Cellar/gcc/11.2.0/include/c++/11.1.0/bits/stl_vector.h:1043:7: note: candidate: 'std::vector<_Tp, _Alloc>::reference std::vector<_Tp,
_Alloc>::operator[](std::vector<_Tp, _Alloc>::size_type) [with _Tp = COLOR; _Alloc = std::allocator<COLOR>; std::vector<_Tp, _Alloc>::reference = COLOR&; std::vector<_Tp,
_Alloc>::size_type = long unsigned int]'
 1043 |       operator[](size_type __n) _GLIBCXX_NOEXCEPT
      |       ^~~~~~~~
/opt/homebrew/Cellar/gcc/11.2.0/include/c++/11.1.0/bits/stl_vector.h:1043:28: note:   no known conversion for argument 1 from 'const std::tuple<long unsigned int, double>' to
'std::vector<COLOR>::size_type' {aka 'long unsigned int'}
 1043 |       operator[](size_type __n) _GLIBCXX_NOEXCEPT
      |                  ~~~~~~~~~~^~~
/opt/homebrew/Cellar/gcc/11.2.0/include/c++/11.1.0/bits/stl_vector.h:1061:7: note: candidate: 'std::vector<_Tp, _Alloc>::const_reference std::vector<_Tp,
_Alloc>::operator[](std::vector<_Tp, _Alloc>::size_type) const [with _Tp = COLOR; _Alloc = std::allocator<COLOR>; std::vector<_Tp, _Alloc>::const_reference = const COLOR&;
std::vector<_Tp, _Alloc>::size_type = long unsigned int]'
 1061 |       operator[](size_type __n) const _GLIBCXX_NOEXCEPT
      |       ^~~~~~~~
/opt/homebrew/Cellar/gcc/11.2.0/include/c++/11.1.0/bits/stl_vector.h:1061:28: note:   no known conversion for argument 1 from 'const std::tuple<long unsigned int, double>' to
'std::vector<COLOR>::size_type' {aka 'long unsigned int'}
 1061 |       operator[](size_type __n) const _GLIBCXX_NOEXCEPT
      |                  ~~~~~~~~~~^~~
In file included from bfs_eg.cpp:5:
bfs.hpp:38:15: error: no matching function for call to 'std::queue<long unsigned int>::push(const std::tuple<long unsigned int, double>&)'
   38 |         Q.push(v);
```

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>>;
};
```

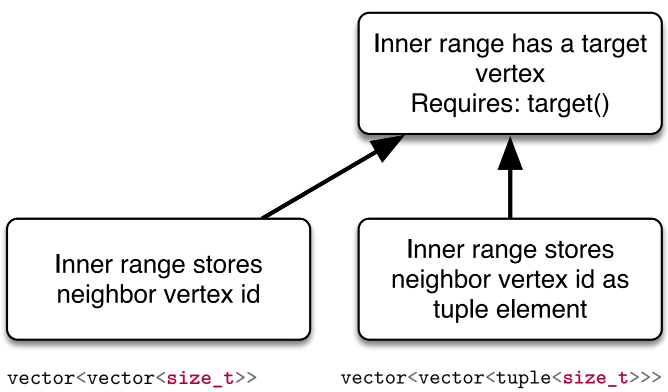
Type of vertex identifier (cannot be inferred)

target() is a customization-point object (CPO)

Requirements: BFS Algorithm

```
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

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

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

```
using Graph = std::vector<vector<std::tuple<size_t, size_t>>>;

template <iclass U>
auto tag_invoke(const target_tag, const Graph& graph, const U& e) {
  return std::get<0>(e);
}
```

```
using Graph = nw::graph::compressed_sparse<double>;

template <class U>
auto tag_invoke(const target_tag, const Graph& graph, const U& e) {
  return std::get<0>(e);  /* implementation defined */
}
```

Don’t need an actual container of containers

Minimalist App concept ored Property Graph

```
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;
}
```

Property is stored on the edge

Minimalist Approach: Index Property Graph

```
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 we
      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

```
template <adjacency_list_graph Graph, class WeightFunction>
auto dijkstra(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

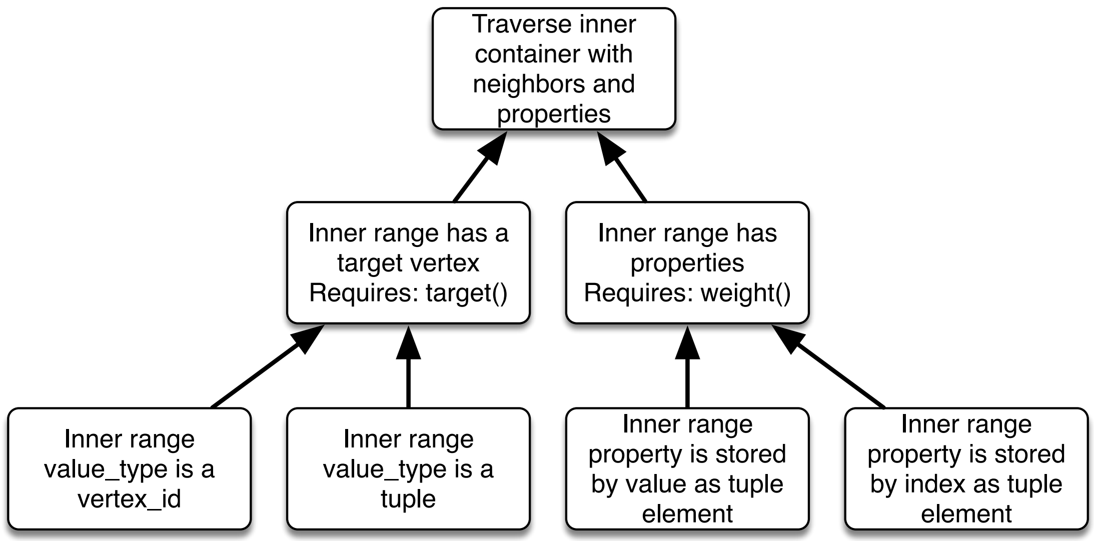
Lookup stored property

```
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)]); });
```

Lookup property from table

Lifted Dijkstra



Requirements: Dijkstra Algorithm

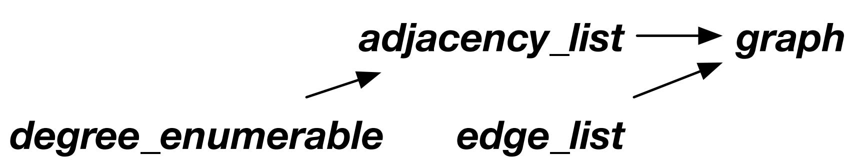
```
template <adjacency_list_graph Graph, class WeightFunction>
auto dijkstra(const Graph& graph, vertex_id_t<Graph> source, WeightFunction weights);
```

- `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

```
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

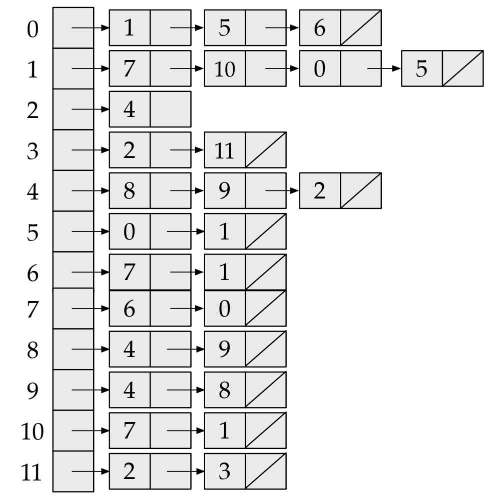
Might want to compute predecessors (paths)

Might want to compute something else

Range Adaptors

```
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::end;
}
```

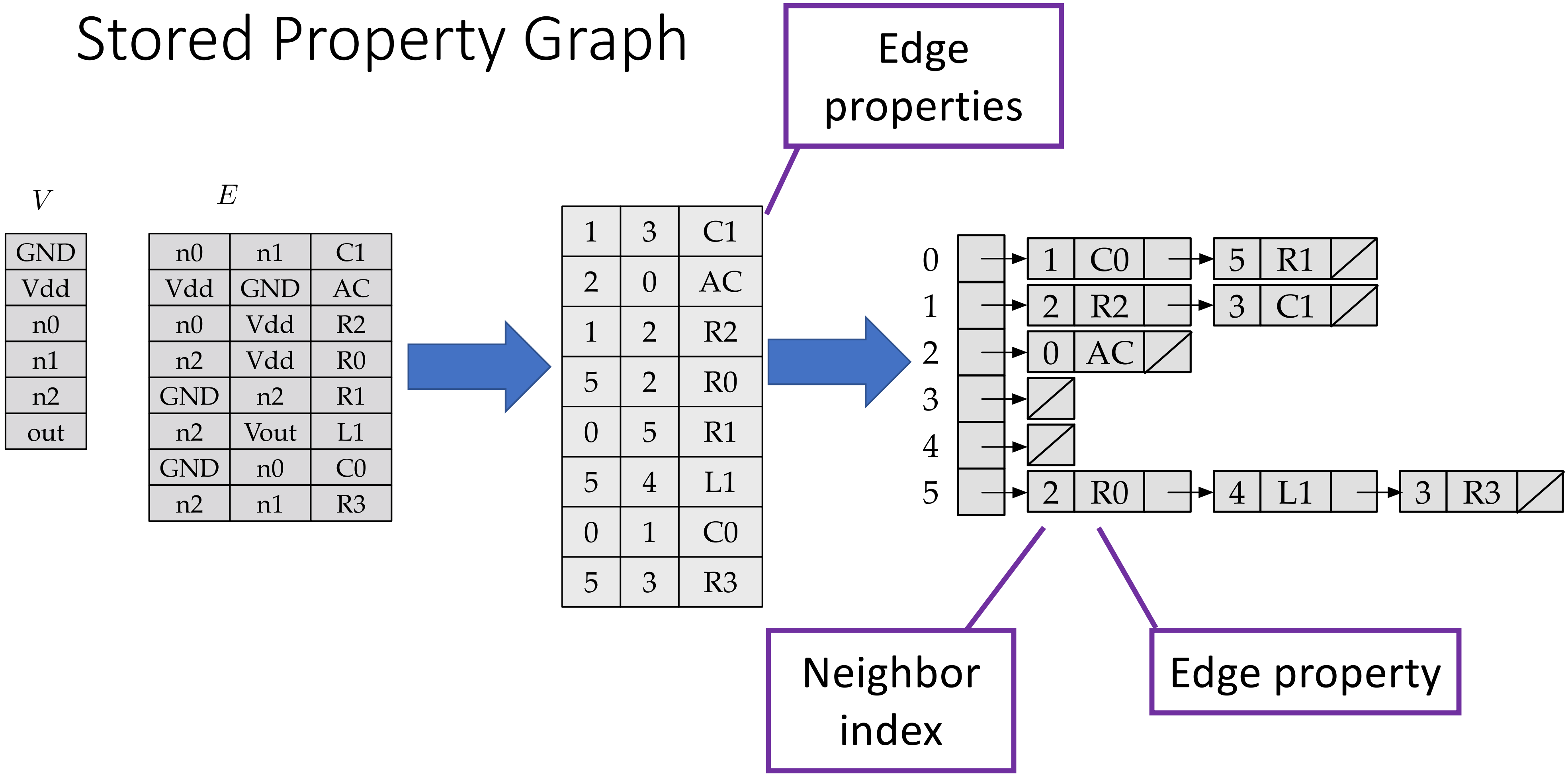
Index Adjacency List



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



Index Property Graph

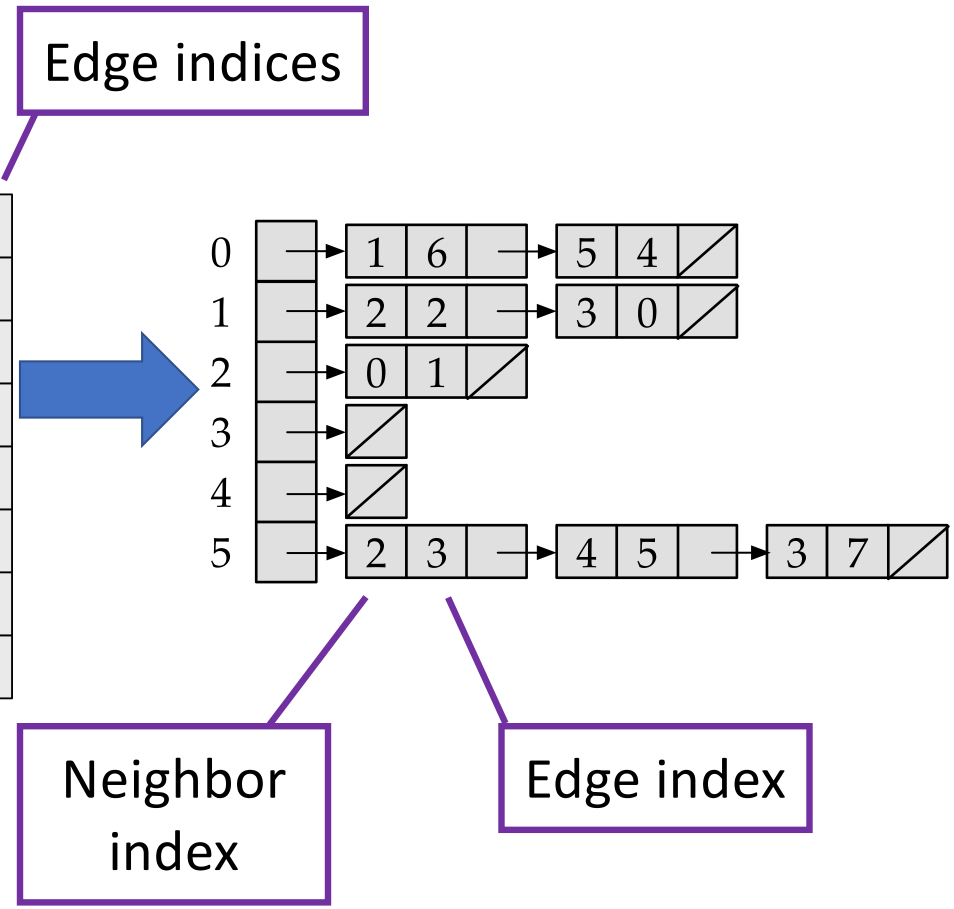
V

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

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



Graph Construction (Library Functions)

```
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 i

template <std::ranges::random_access_range V, std::ranges::random_access_range E,
          adjacency_list Graph = std::vector<std::vector<std::tuple<size_t, size_t>>>>
auto make_index_graph(const V& vertices, const E& edges, bool direct                  i

template <std::ranges::random_access_range V, std::ranges::forward_r
          adjacency_list Graph = std::vector<std::vector<
              decltype(std::tuple_cat(std::make_tuple(size_t{}), pro                  )
auto make_property_graph(const V& vertices, const E& edges, bool di
```

```
auto G = make_property_graph(ospf_vertices, ospf
```

```
std::vector<std::string>
  ospf_vertices {
  "RT1",
  "RT2",
  "RT3",
  "RT4",
  "RT5",
  "RT6",
  "RT7",
  "RT8",
  "RT9",
  "RT10",
  "RT11",
  "RT12",
  "N1",
  "N2",
  "N3",
  "N4",
  "N6",
  "N7",
  "N8",
  "N9",
  "N10",
  "N11",
  "N12",
 "N13",
 "N14",
 "N15",
 "H1",
};
```

```
std::vector<std::tuple<
  std::string, std::string,
  size_t>> ospf_edges {
  {"RT1", "N1", 3},
  {"RT1", "N3", 1},
  {"RT2", "N2", 3},
  {"RT2", "N3", 1},
  {"RT3", "RT6", 8},
  {"RT3", "N3", 1},
  {"RT3", "N4", 2},
  {"RT4", "N3", 1},
  {"RT4", "RT5", 8},
  {"RT5", "RT4", 8},
  {"RT5", "RT6", 7},
  {"RT5", "RT7", 6},
  {"RT5", "N12", 8},
  {"RT5", "N13", 8},
  {"RT5", "N14", 8},
  {"RT6", "RT3", 6},
  {"RT6", "RT5", 6},
  {"RT6", "RT10", 7},
  {"RT7", "RT5", 6},
  {"RT7", "N6", 1},
  {"RT7", "N12", 2},
  {"RT7", "N15", 9},
  {"RT8", "N6", 1},
  {"RT8", "N7", 4},
  {"RT9", "N9", 1},
  {"RT9", "N11", 3},
  {"RT10", "RT6", 5},
  {"RT10", "N6", 1},
  {"RT10", "N8", 3},
  {"RT11", "N8", 2},
  {"RT11", "N9", 1},
  {"RT12", "N9", 1},
  {"RT12", "N10", 2},
  {"RT12", "H1", 10},
  {"N3", "RT1", 0},
  {"N3", "RT2", 0},
  {"N3", "RT3", 0},
  {"N3", "RT4", 0},
  {"N6", "RT7", 0},
  {"N6", "RT8", 0},
  {"N6", "RT10", 0},
  {"N8", "RT10", 0},
  {"N8", "RT11", 0},
  {"N9", "RT9", 0},
  {"N9", "RT11", 0},
  {"N9", "RT12", 0},
  };
```

Compressed Graph Type

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

```
template <int idx, std::unsigned_integral edge_idx, std::u        ral vertex_idx,
          typename... Attributes>
class index_adjacency;
```

Index

V

SEA
MSP
SLC
DTW
ATL
BOS

E

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

Index edge list

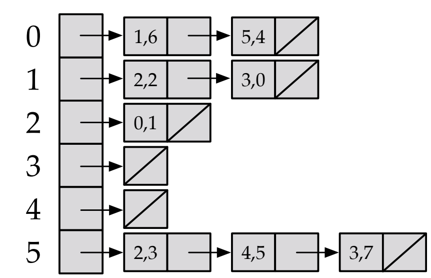
1	3	0
2	0	1
1	2	2
5	2	3
0	5	4
5	4	5
0	1	6
5	3	7

Compressed

0	0
1	2
2	4
3	5
4	5
5	5
	8

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

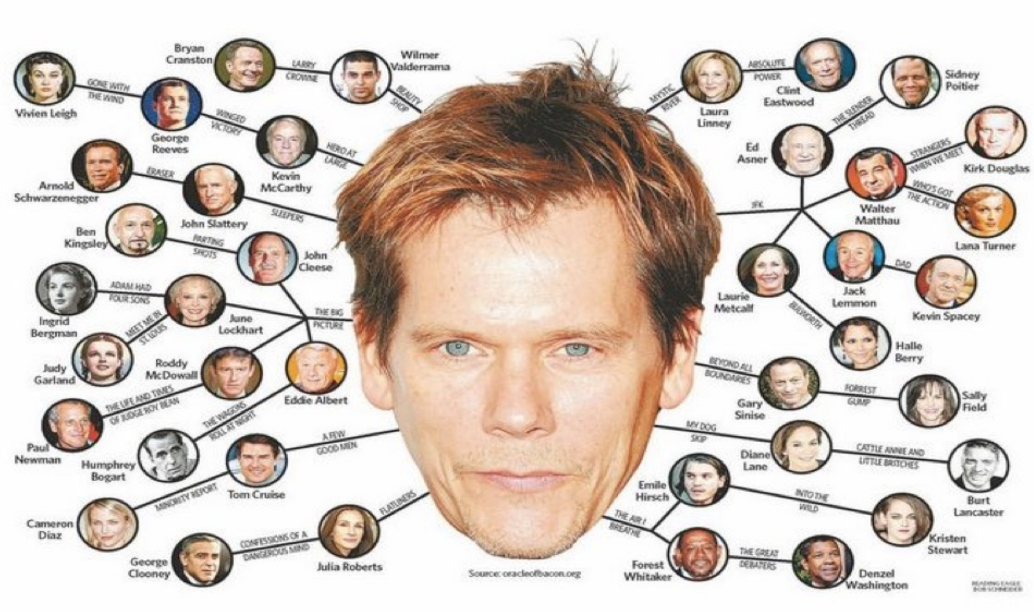
Storage



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

```
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

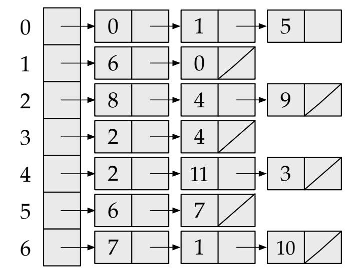
size = 18

Index
Edge List

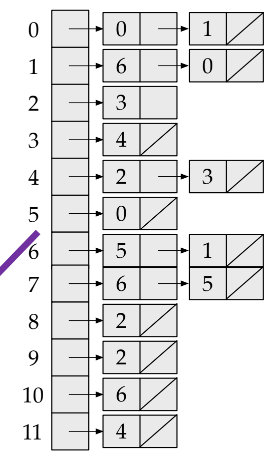
0	0
0	1
0	5
6	7
6	1
1	0
1	6
5	7
5	6
2	8
2	4
2	9
3	2
3	4
4	3
4	11
4	2
6	10

num_vertices = (7, 12)

Index Adjacency List <0>

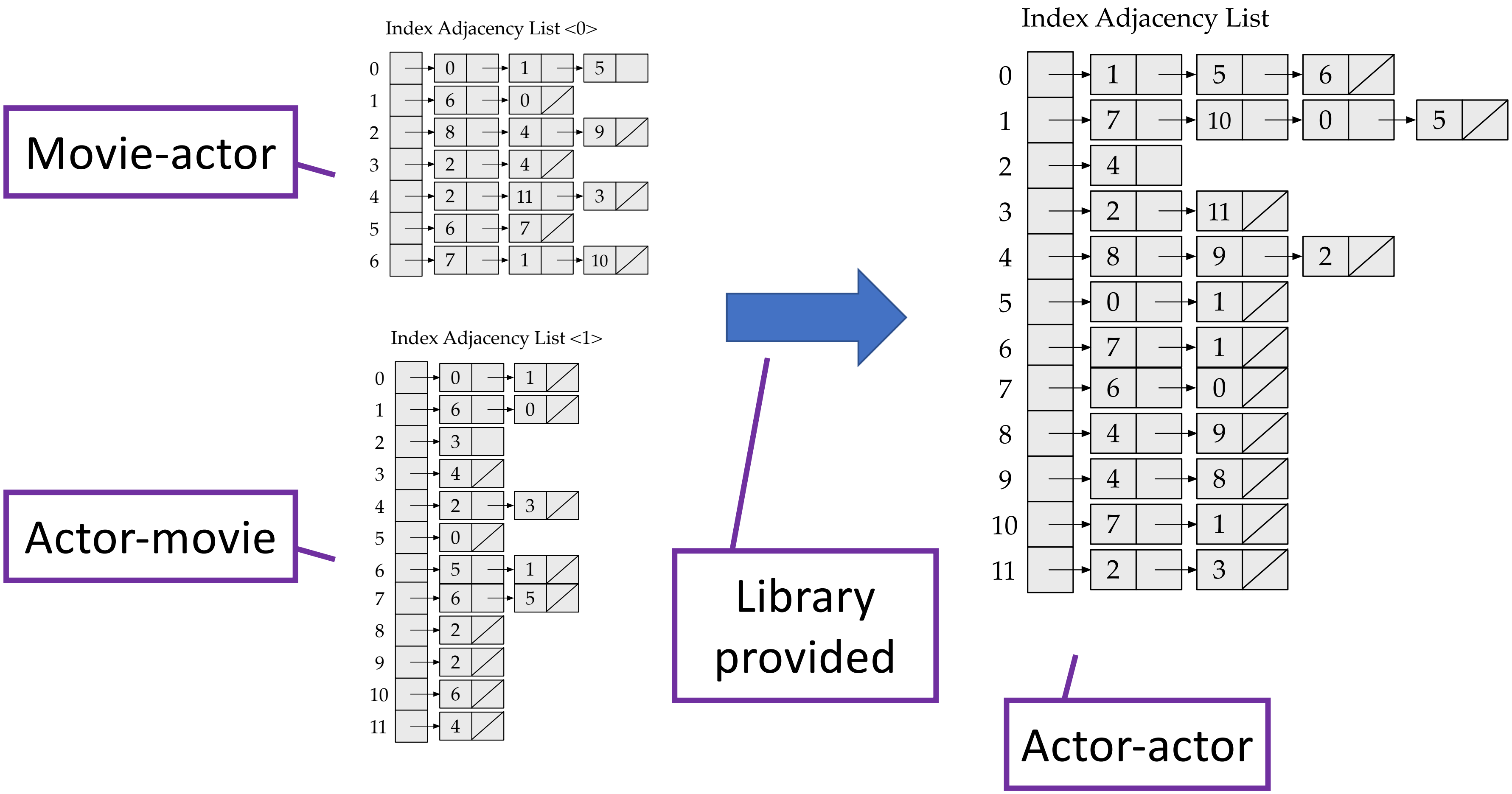


Index Adjacency List <1>



Index each other instead of themselves

Bipartite Graphs



Six Degrees of Kevin Bacon (IMDB version)

```
#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)

```
std::vector<std
  "Tom Cruise",
  "Kevin Bacon"
  "Hugo Weaving
  "Carrie-Anne
  "Natalie Port
  "Jack Nichols
  "Kelly McGill
```

```
std::vector<std::string> m
    "A Few Good Men",
    "Top Gun",
    "Black Swan",
    "V for Vendetta",
    "The Matrix",
    "Witness",
    "What Lies Beneath",
    "Closer",
    "Flatliners",
};
```

```
std::vector<std::tuple<std::string, std::string>>
  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", "Carrie-Anne Moss"},
    {"The Matrix", "Keanu Reeves"},
    {"The Matrix", "Hugo Weaving"},
    {"What Lies Beneath", "Michelle Pfeiffer"},
    {"Closer", "Natalie Portman"},
    {"Closer", "Julia Roberts"},
    {"Flatliners", "Kevin Bacon"},
    {"Flatliners", "Julia Roberts"},
};
```

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

```
Kevin Bacon has a bacon number of 0

Tom Cruise has a bacon number of 1
   Tom Cruise starred with Kevin Bacon in A Few Good Men

Hugo Weaving has a bacon number of 3
   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

Carrie-Anne Moss has a bacon number of 4
   Carrie-Anne Moss starred with Hugo Weaving in The Matrix
   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

Natalie Portman has a bacon number of 2
   Natalie Portman starred with Julia Roberts in Closer
   Julia Roberts starred with Kevin Bacon in Flatliners

Jack Nicholson has a bacon number of 1
   Jack Nicholson starred with Kevin Bacon in A Few Good Men

Kelly McGillis has a bacon number of 2
   Kelly McGillis starred with Tom Cruise in Top Gun
   Tom Cruise starred with Kevin Bacon in A Few Good Men
```

```
Harrison Ford has a bacon number of 1
   Harrison Ford starred with Kevin Bacon in What Lies Beneath

Sebastian Stan has a bacon number of 3
   Sebastian Stan starred with Natalie Portman in Black Swan
   Natalie Portman starred with Julia Roberts in Closer
   Julia Roberts starred with Kevin Bacon in Flatliners

Mila Kunis has a bacon number of 3
   Mila Kunis starred with Natalie Portman in Black Swan
   Natalie Portman starred with Julia Roberts in Closer
   Julia Roberts starred with Kevin Bacon in Flatliners

Michelle Pfeiffer has a bacon number of 1
   Michelle Pfeiffer starred with Kevin Bacon in What Lies Beneath

Keanu Reeves has a bacon number of 4
   Keanu Reeves starred with Hugo Weaving in The Matrix
   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

Julia Roberts has a bacon number of 1
   Julia Roberts starred with Kevin Bacon in Flatliners
```

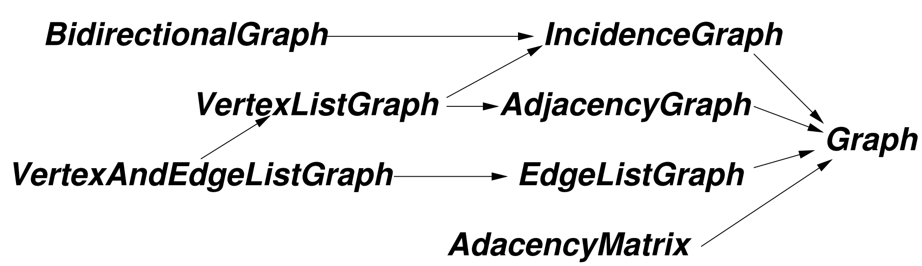
Boost Graph Library

Algorithm	BGL Input Requirement on Graph Concept	Other Requirements
BC clustering	Vertex List Graph and Incidence Graph (and Edge List Graph and Mutable Graph)	
bellman ford	Edge List Graph (and Vertex List Graph)	directed or undirected
bfs	Vertex List Graph and Incidence Graph	directed or undirected
bicc	Vertex List Graph and Incidence Graph	undirected
BK Max Flow	Vertex List Graph and Incidence Graph and Edge List Graph	directed
Brandes BC	Vertex List Graph and Incidence Graph	
CC	Vertex List Graph and Incidence Graph	undirected
cuthill mckee	Incidence Graph	undirected
delta stepping	Vertex List Graph and Incidence Graph	directed or undirected
dfs	Vertex List Graph and Incidence Graph	[illegible]
dijkstra	Vertex List Graph and Incidence Graph	directed or undirected, weighted
dominator tree	Vertex List Graph and Bidirectional Graph	directed (?)
EK Max Flow	Vertex List Graph and Incidence Graph	directed
floyd warshall	Vertex List Graph (and Vertex And Edge List Graph)	directed or undirected
johnson all pairs	Vertex List Graphand Incidence graph and Edge List Graph	directed or undirected
Kruskal	Vertex List Graph and Edge List Graph	undirected
max card matching	Vertex And Edge List Graph and Incidence Graph	undirected
max weighted matching	Vertex And Edge List Graph and Incidence Graph	undirected
min degree ordering	Vertex List Graph and Incidence Graph and Adjacency Graph and Mutable Graph	directed
page rank	Vertex List Graph and Incidence Graph	directed
PR Max Flow	Vertex List Graph	directed
Prim	Vertex List Graph and Incidence Graph	undirected
RCM	Incidence Graph	undirected
SCC	Vertex List Graph and Incidence Graph	directed
stoer wagner min cut	Vertex List Graph and Incidence Graph	undirected
topological sort	Vertex List Graph and Incidence Graph	dag
transitive closure	Vertex List Graph and Adjacency Graph and Adjacency Matrix	directed
triangle counting	Vertex List Graph and Incidence Graph	undirected

Vertex List Graph and Incidence Graph

Boost Graph Library

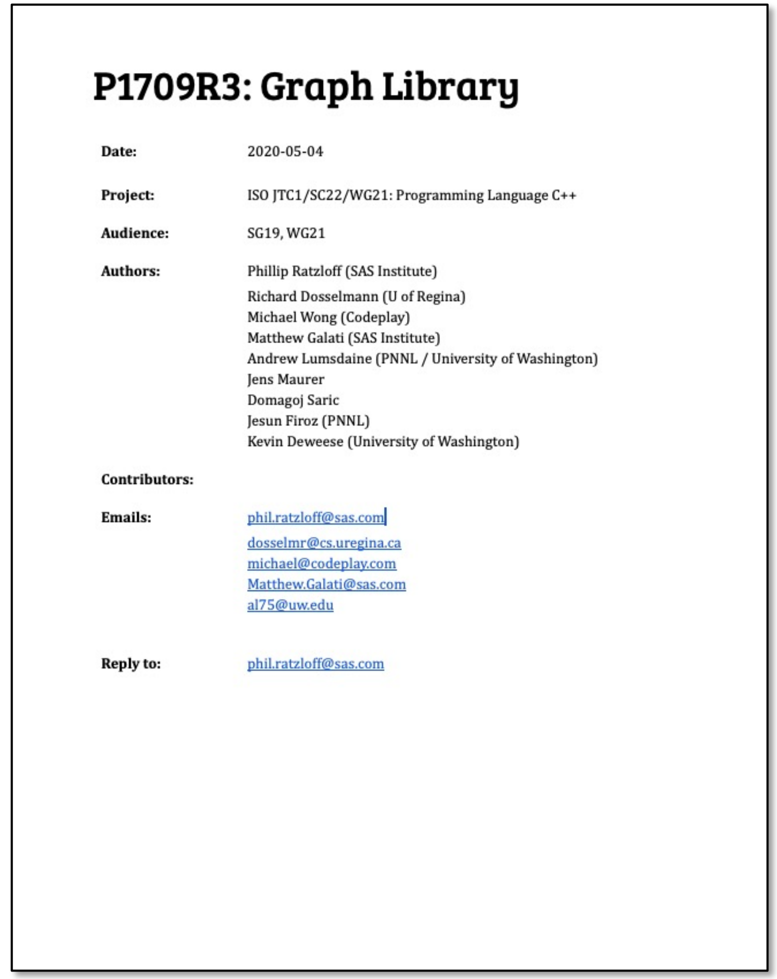
- The BGL concepts were derived with same generic programming process
- Our concepts cover the same space
- But has streamlined concepts (rather than Swiss Army Knife)



BGL Concept	Purpose	Functions	Other
Adjacency Graph	Iterate neighbors of vertex	adjacent_vertices(v, g)	
Adjacency Matrix	Random access to edge	edge(u, v, g)	
BiDirectional Graph	Iterate over in edges	in_edges(u, g) in_degree(u, g)	
Edge List Graph	Iterate all edges in graph	edges(g), num_edges(g)	source(e, g), target(e, g)
Incidence Graph	Access neighbors of vertex, including edges	out_edges(u, g) out_degree(u, g)	source(e, g), target(e, g)
Vertex And Edge List Graph	Refines Vertex List Graph and Edge List Graph		
Vertex List Graph	Iterate all vertices in graph	vertices(g), num_vertices(g)	

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!



P1709R3: Graph Library

Date:	2020-05-04
Project:	ISO JTC1/SC22/WG21: Programming Language C++
Audience:	SG19, WG21
Authors:	Phillip Ratzloff (SAS Institute) Richard Dosselmann (U of Regina) Michael Wong (Codeplay) Matthew Galati (SAS Institute) Andrew Lumsdaine (PNNL / University of Washington) Jens Maurer Domagoj Saric Jesun Firoz (PNNL) Kevin Deweese (University of Washington)
Contributors:	
Emails:	phil.ratzloff@sas.com dosselmr@cs.uregina.ca michael@codeplay.com Matthew.Galati@sas.com al75@uw.edu
Reply to:	phil.ratzloff@sas.com

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

```
<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

```
<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)

Raw, c-like

```
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]; }}
```

Iterator

```
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

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

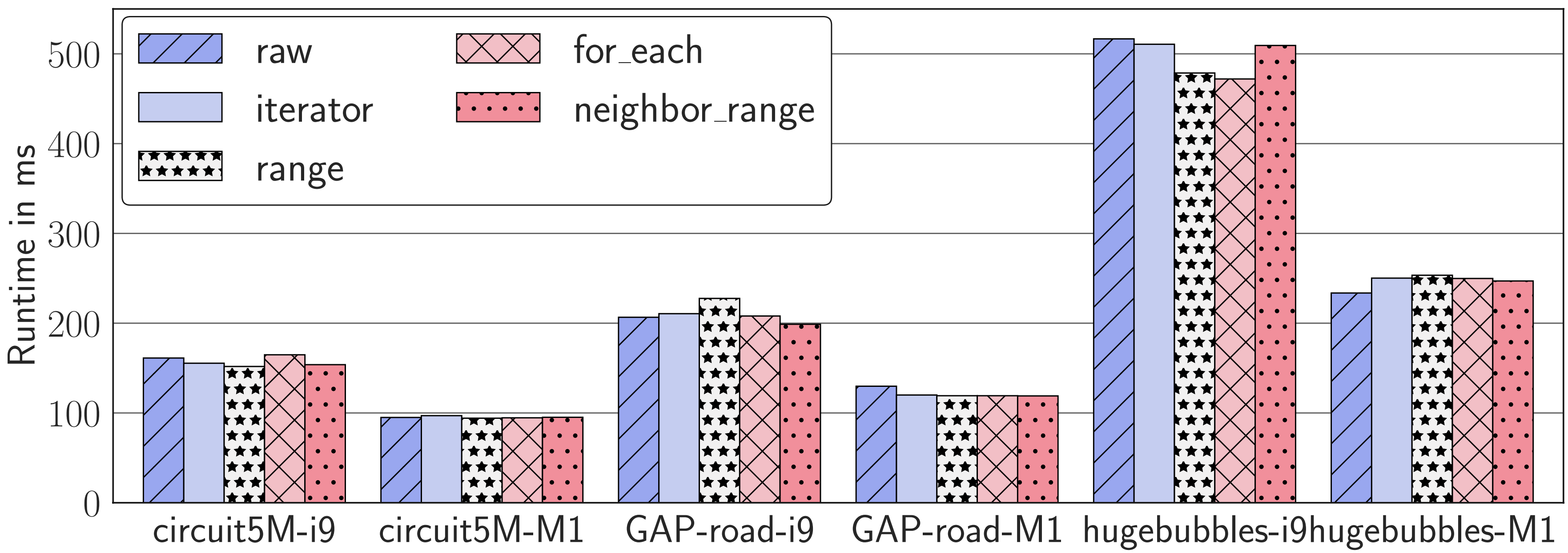
Abstraction Penalty

- Continued…

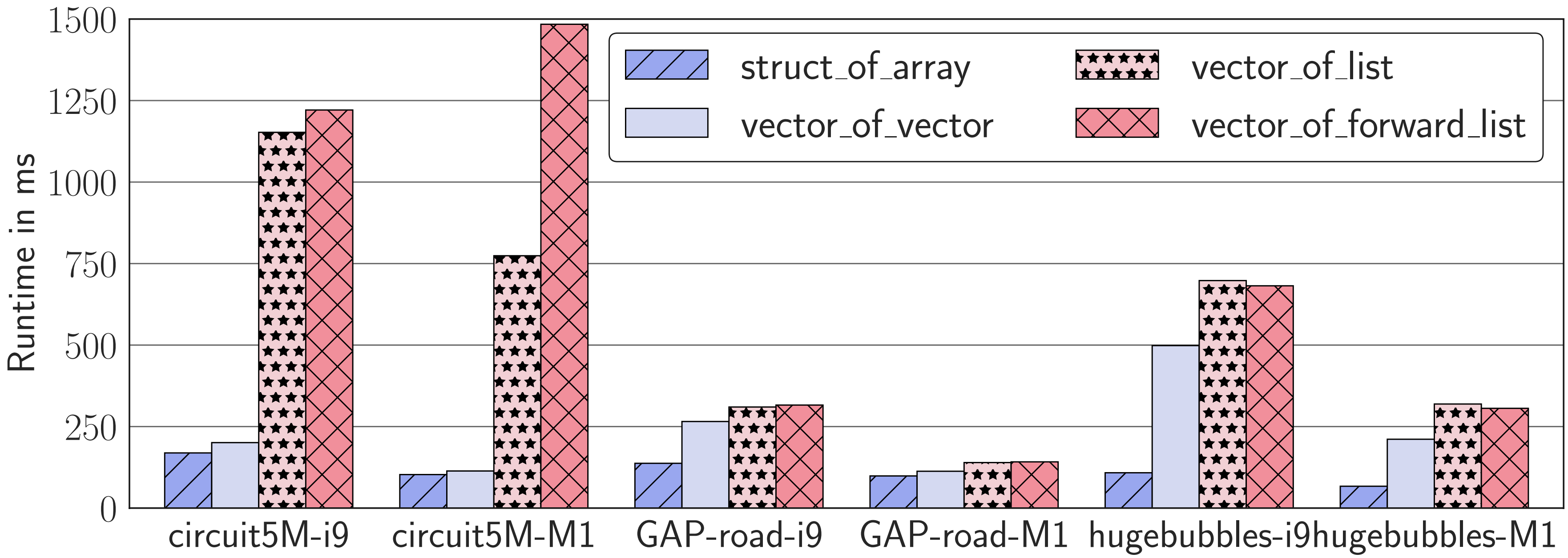
```
auto per = make_edge_range<0>(graph);
for (auto&& j : per) {
  y[std::get<0>(j)] += x[std::get<1>(j)] * std::get<2>(j); }
```

```
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); });
```

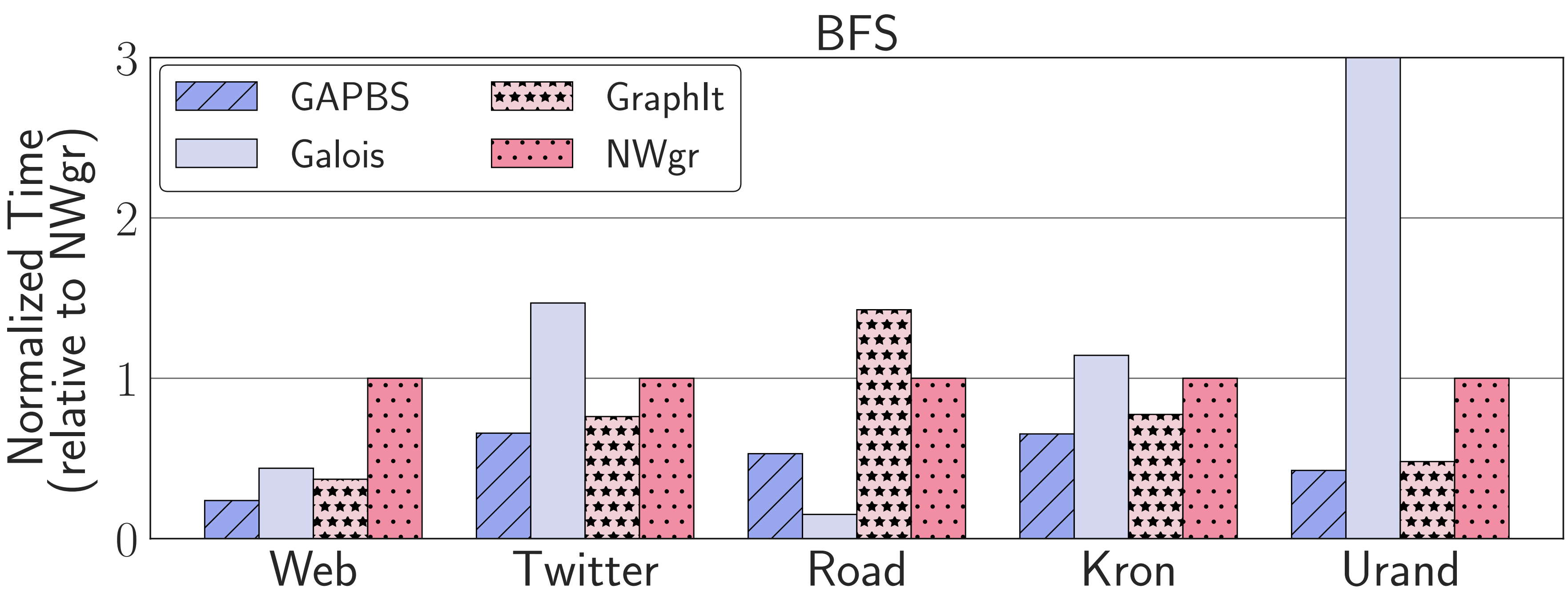
Abstraction penalty



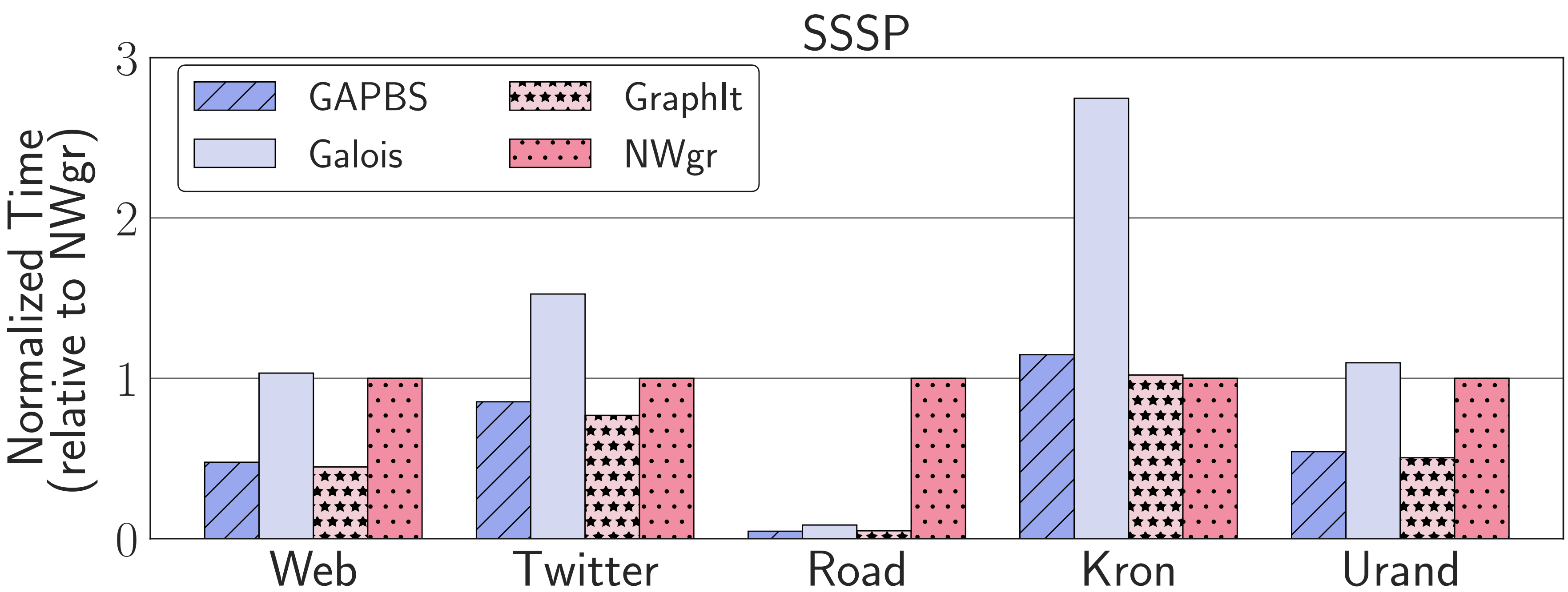
Abstraction Penalty



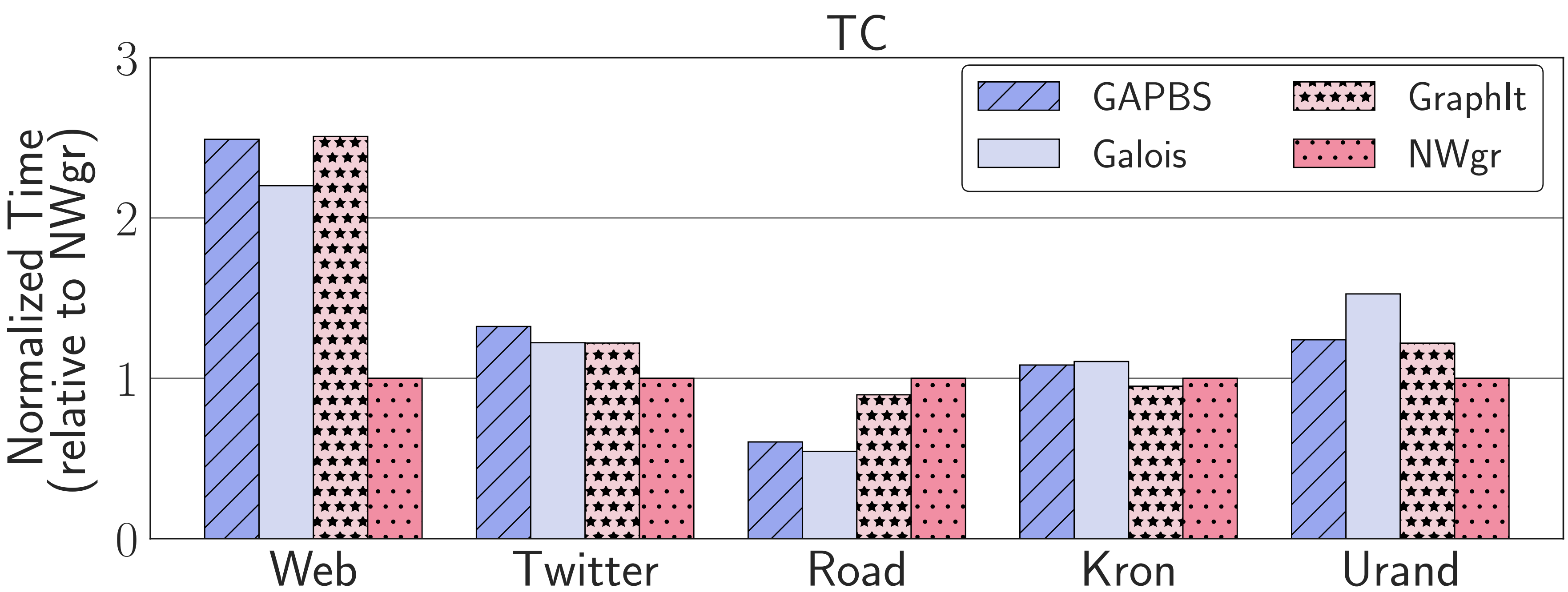
Performance Comparisons



Performance Comparisons



Performance Comparisons



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