Graph Algorithms and Data Structures in C++20

Toward std::graph

PHIL RATZLOFF & ANDREW LUMSDAINE

Cppcon
The C++ Conference
2022
September 12th-16th

Phil Ratzloff

Distinguished Software Developer, SAS Institute

Phil is a C++ advocate at SAS Institute. He has used C++ for 27 years on applications using embedded graphs for Cost and Profitability Analysis, and entity resolution.

Andrew Lumsdaine

Principal Software Engineer, TileDB, Inc

Laboratory Fellow, Pacific Northwest National Lab

Affiliate Professor, University of Washington

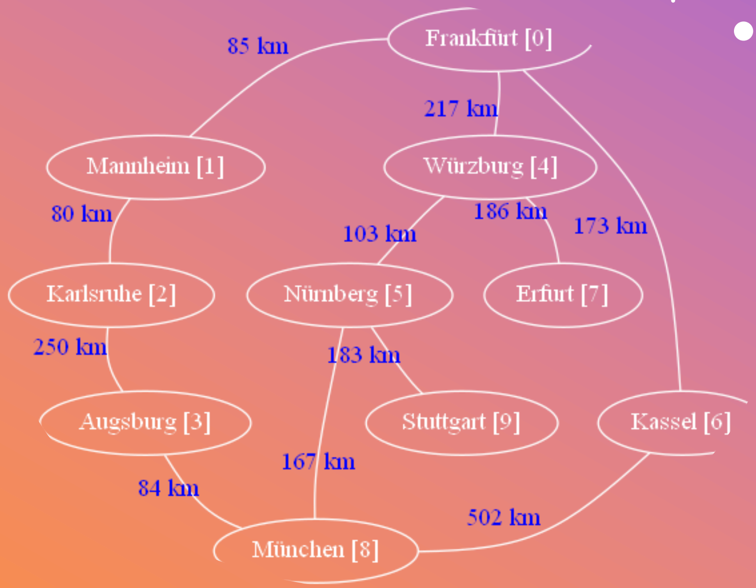
Andrew has worked in many areas related to high-performance computing, including systems, software libraries, and large-scale graph analytics. Open-source software projects resulting from his work include Boost.Graph and Open MPI.

Thanks also to Jesun Firoz, Tony Liu, Kevin Deweese, Scott McMillan, Haley Riggs, Richard Dosselman, Matthew Galati, Muhammad Osama and SG19 Machine Learning

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



AGENDA

Introduction

What Is a Graph?

Example

Algorithms

Views

csr_graph Graph Container

Other Graph Containers

Wrap-Up

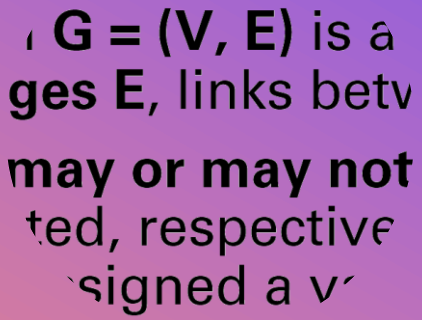
Introduction

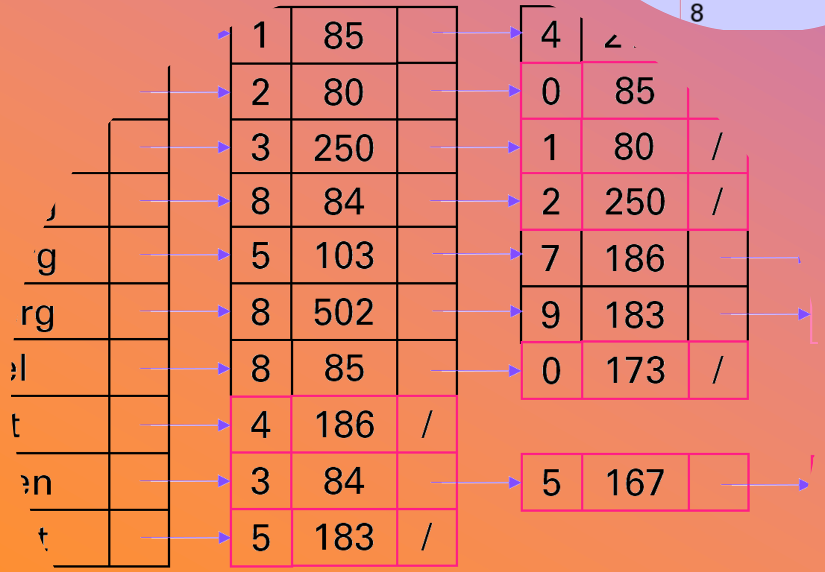
- **~ 2001**
  - boost::graph released
- **2017**
  - NWGraph started
- **2019**
  - SG19: “A graph library would be nice to have”
  - P1709r0 published
  - graph-v1 started
  - Phil & Andrew meet
- **2022**
  - graph-v2 started
  - NWGraph published
  - graph-v2 + NWGraph
  - P1709r3 published?
- **2025**
  - Accepted into Standard?

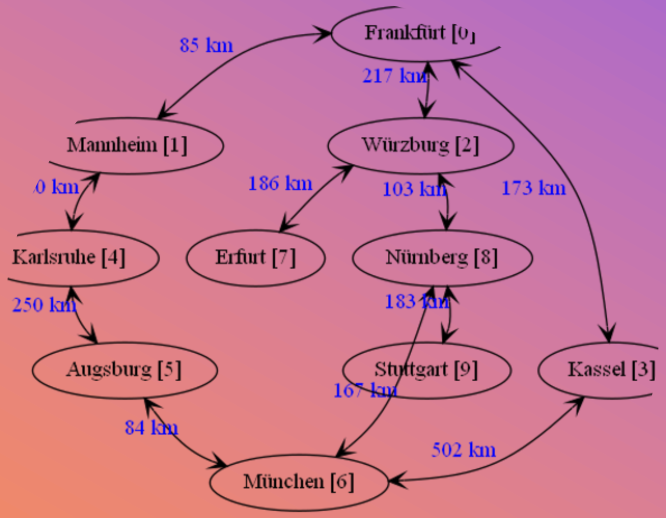
Caveats

- Incomplete
- Work in Progress
- Breaking Changes Expected
- Hasn’t been reviewed outside of SG19 Machine Learning
- There are no guarantees if/when it will be accepted into the standard library

TOWARD STD::GRAPH







WHAT IS A GRAPH?

Adjacency List

Edge List

What Is a Graph?

A graph **G = (V, E)** is a set of **vertices V**, points in a space, and **edges E**, links between these vertices.

Edges may or may not be oriented, that is, directed or undirected, respectively. Moreover, edges may be **weighted**, that is, assigned a value.

Handbook of Graph Theory, Gross, Jonathan L. and Yellen, Jay, CRS Press 2004

What Is a Graph?

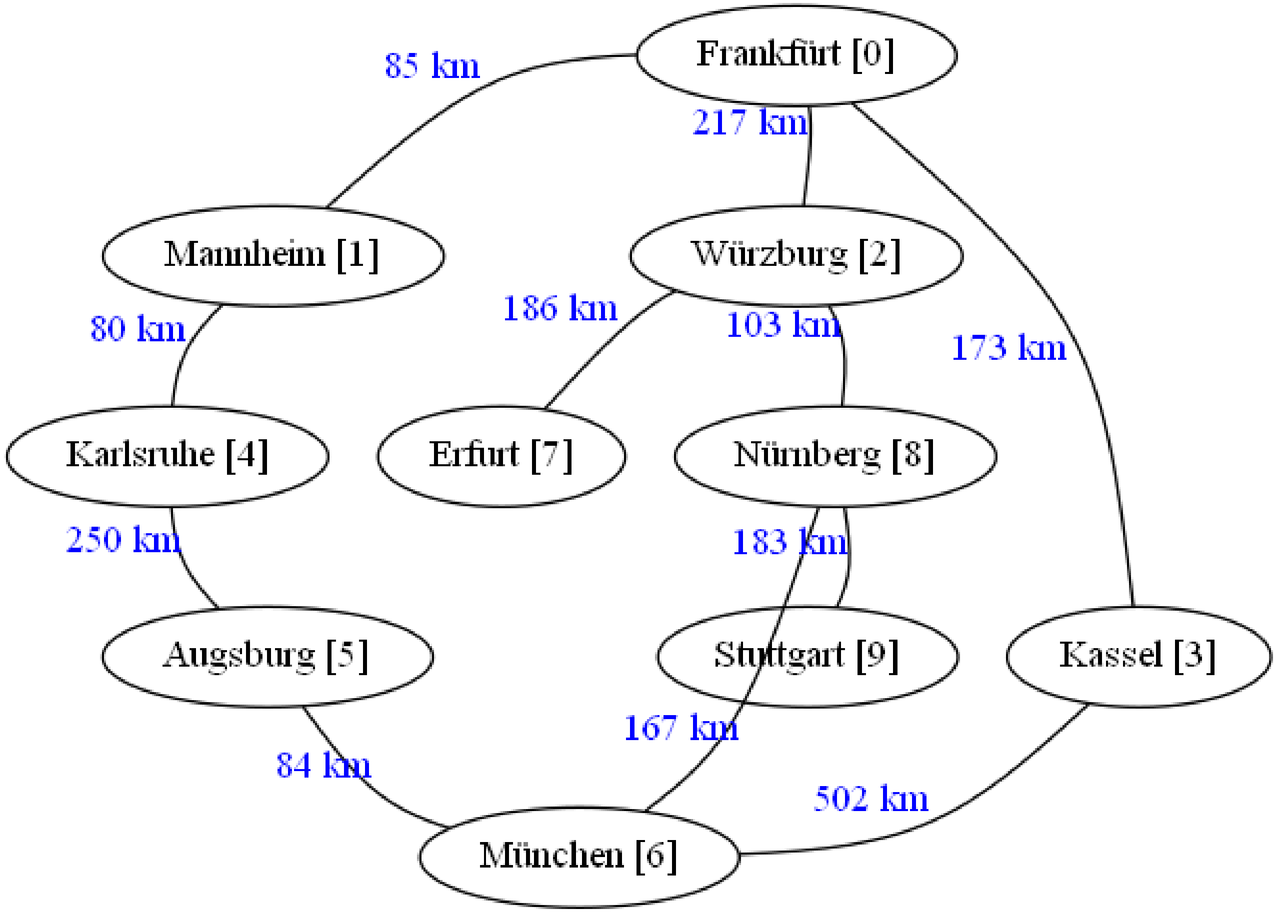
A graph **G = (V, E)** is a set of **vertices V**, points in a space, and **edges E**, links between these vertices.

Edges may or may not be oriented, that is, directed or undirected, respectively. Moreover, edges may be **weighted**, that is, assigned a value.

Vertices and the Graph may also be assigned a value – Phil

Handbook of Graph Theory, Gross, Jonathan L. and Yellen, Jay, CRS Press 2004

What Is a Graph?



Raw Data

From	To	Distance
Frankfürt	Mannheim	85
Frankfürt	Würzburg	217
Frankfürt	Kassel	173
Mannheim	Karlsruhe	80
Karlsruhe	Augsburg	250
Augsburg	München	84
Würzburg	Erfurt	186
Würzburg	Nürnberg	103
Nürnberg	Stuttgart	183
Nürnberg	München	167
Kassel	München	502

Adjacency List

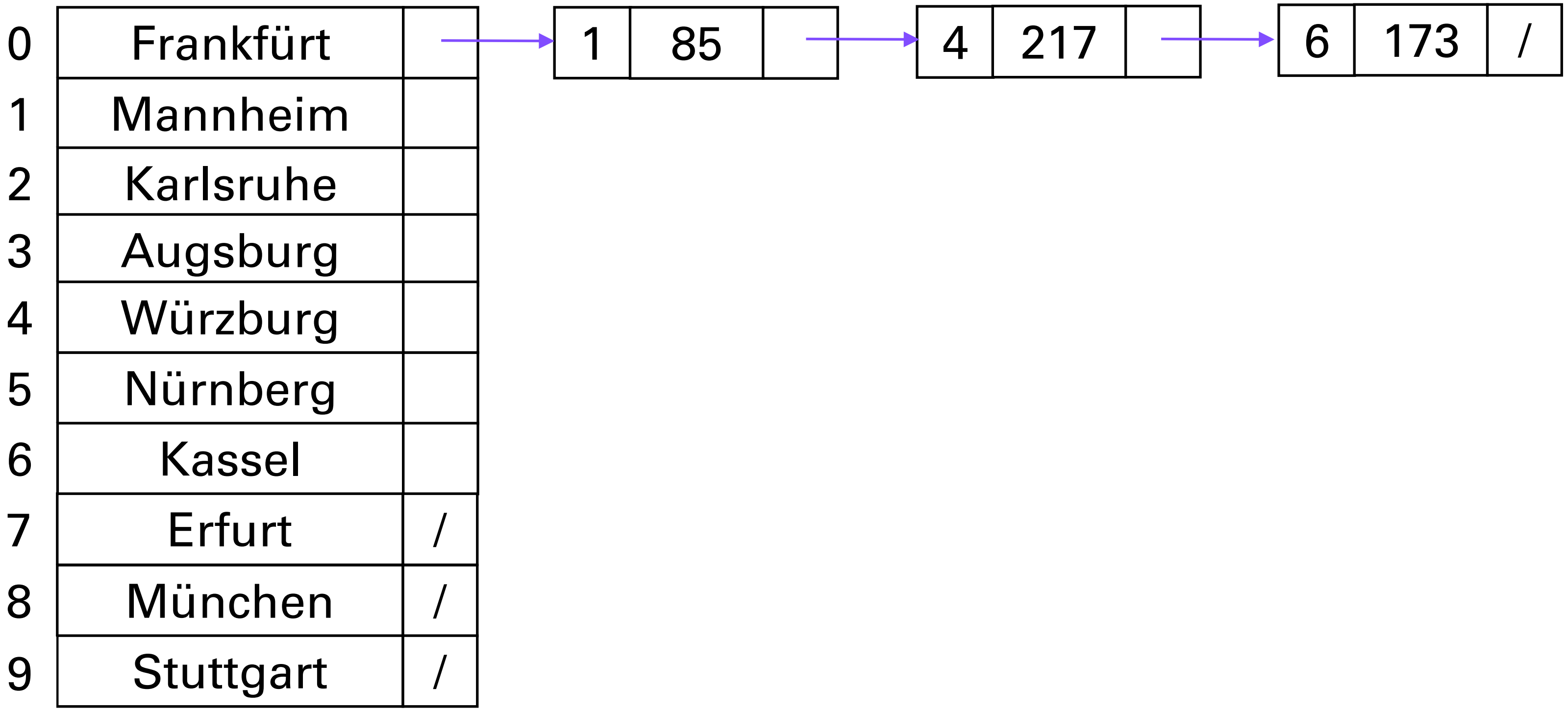
Outer Range - Vertices

0	Frankfürt	
1	Mannheim	
2	Karlsruhe	
3	Augsburg	
4	Würzburg	
5	Nürnberg	
6	Kassel	
7	Erfurt	/
8	München	/
9	Stuttgart	/

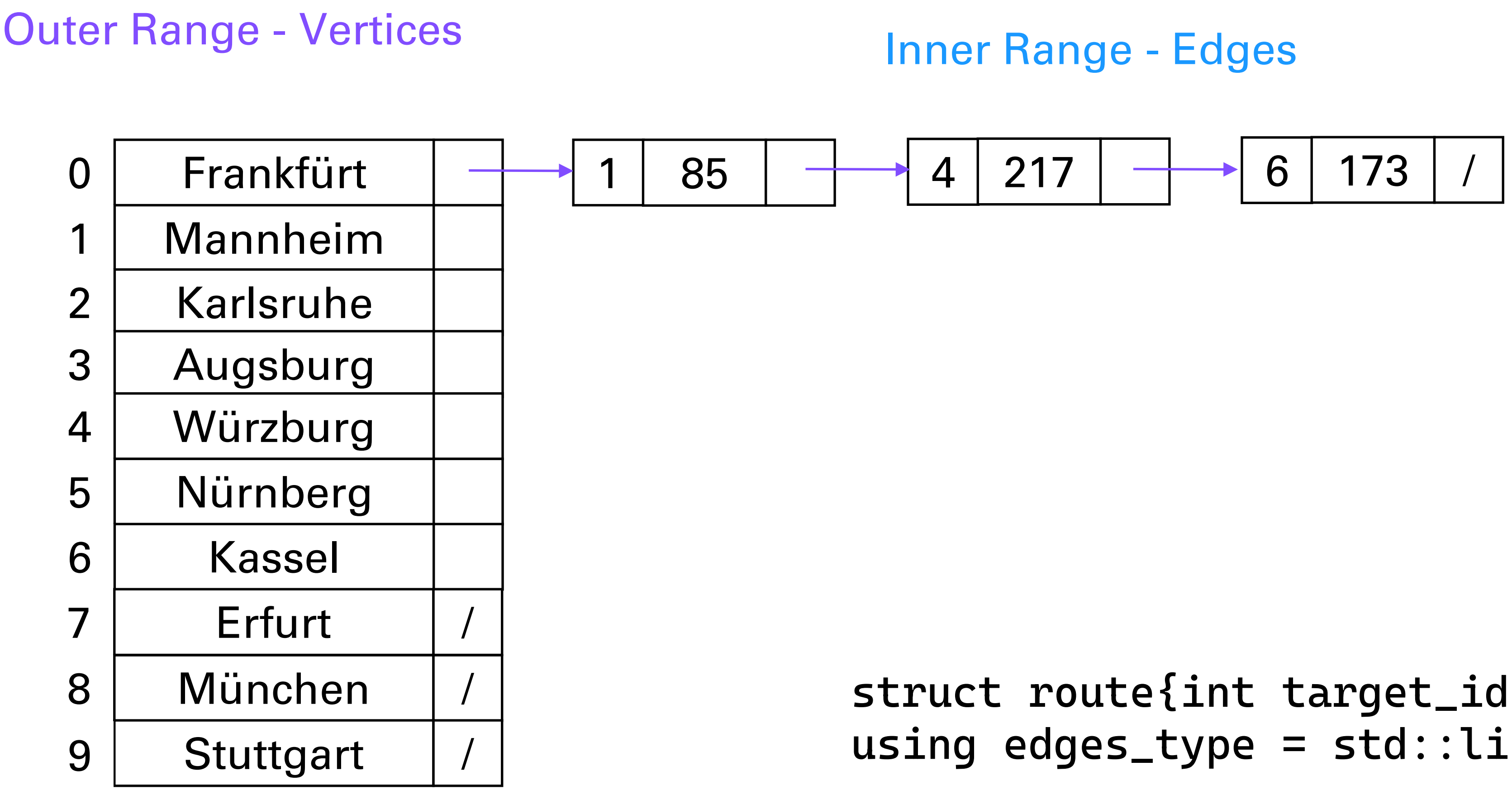
Adjacency List

Outer Range - Vertices

Inner Range - Edges



Adjacency List

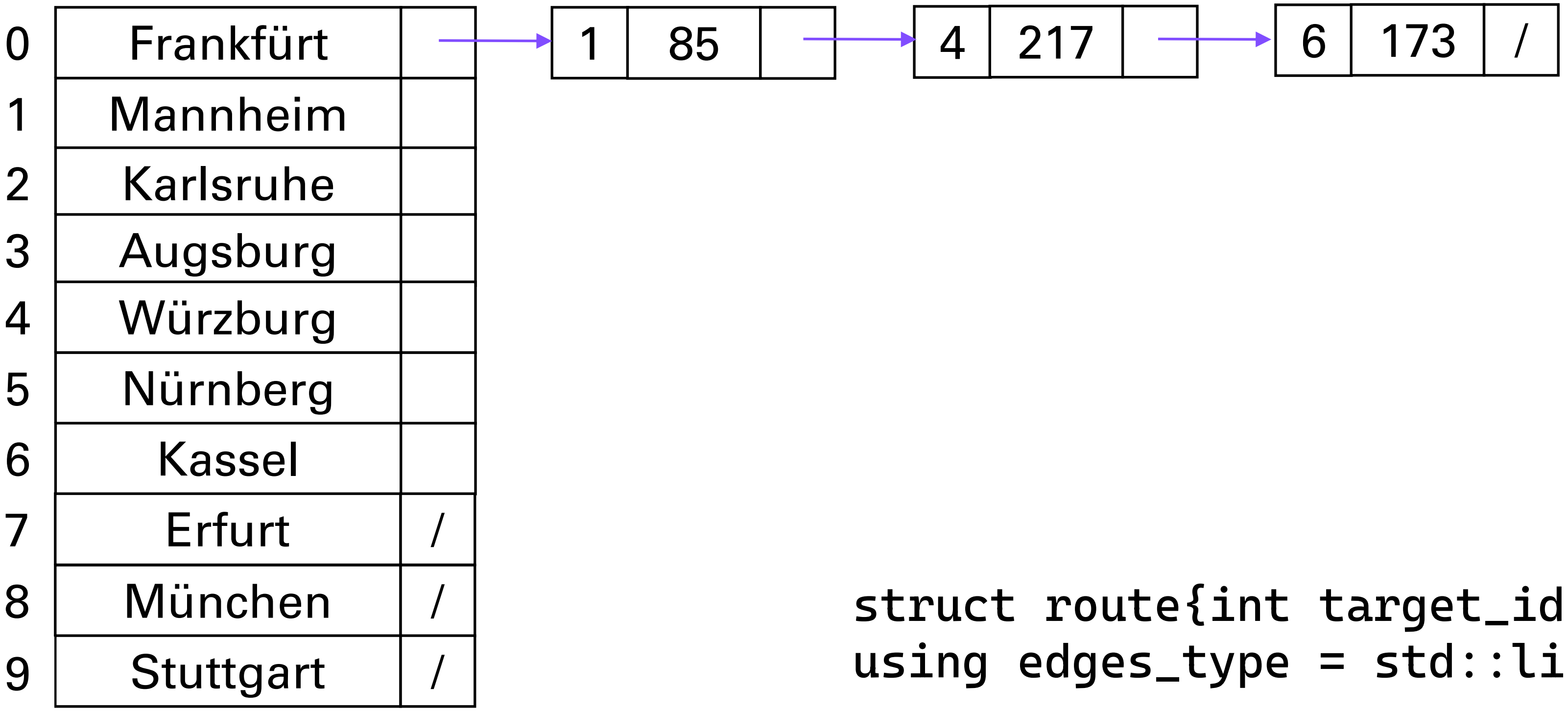



```
struct route{int target_id; double distance;}
using edges_type = std::list<route>;
```

Adjacency List

Outer Range - Vertices

Inner Range - Edges



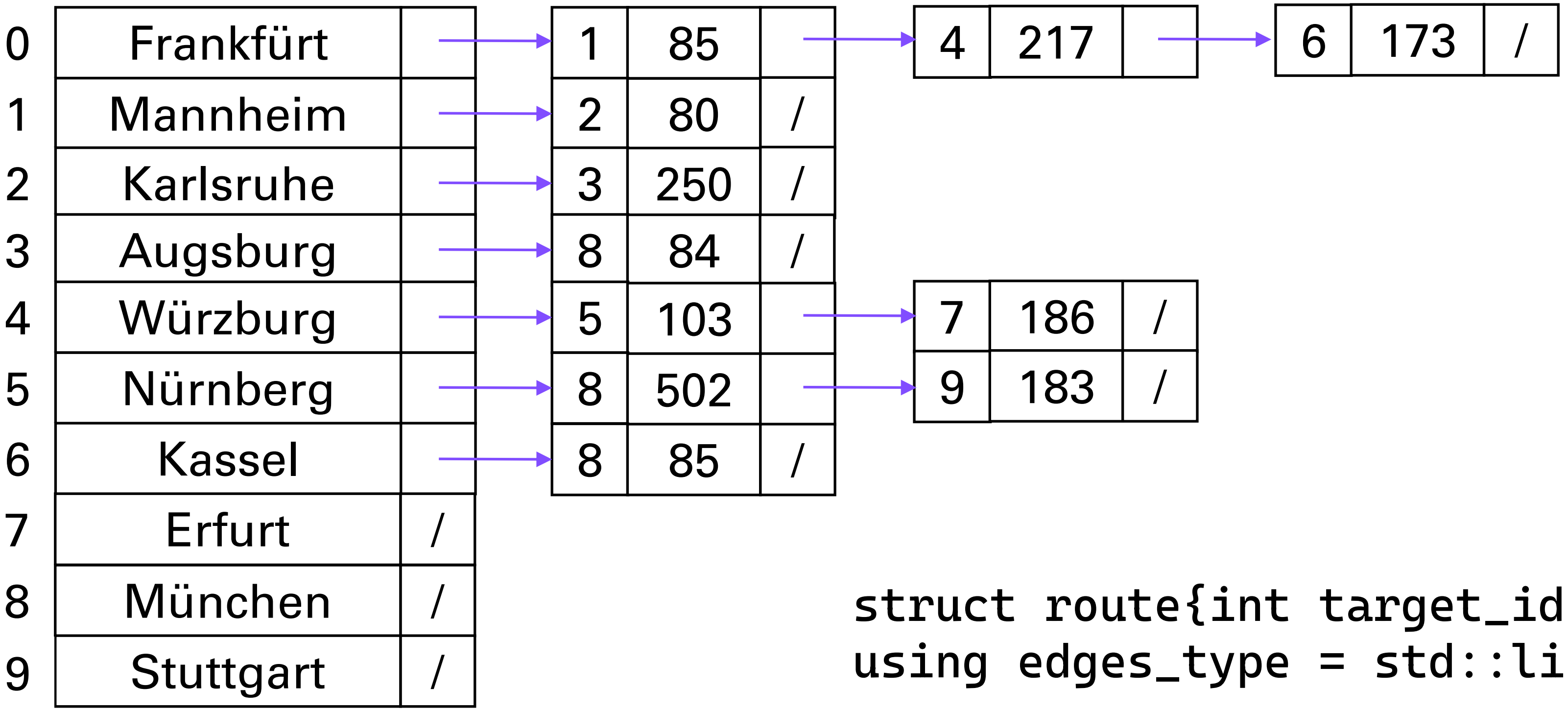

```
struct route{int target_id; double distance;}
using edges_type = std::list<route>;

struct vertex{edges_type edges; string name;}
using vertices_type = std::vector<vertex>;
```

Adjacency List

Outer Range - Vertices

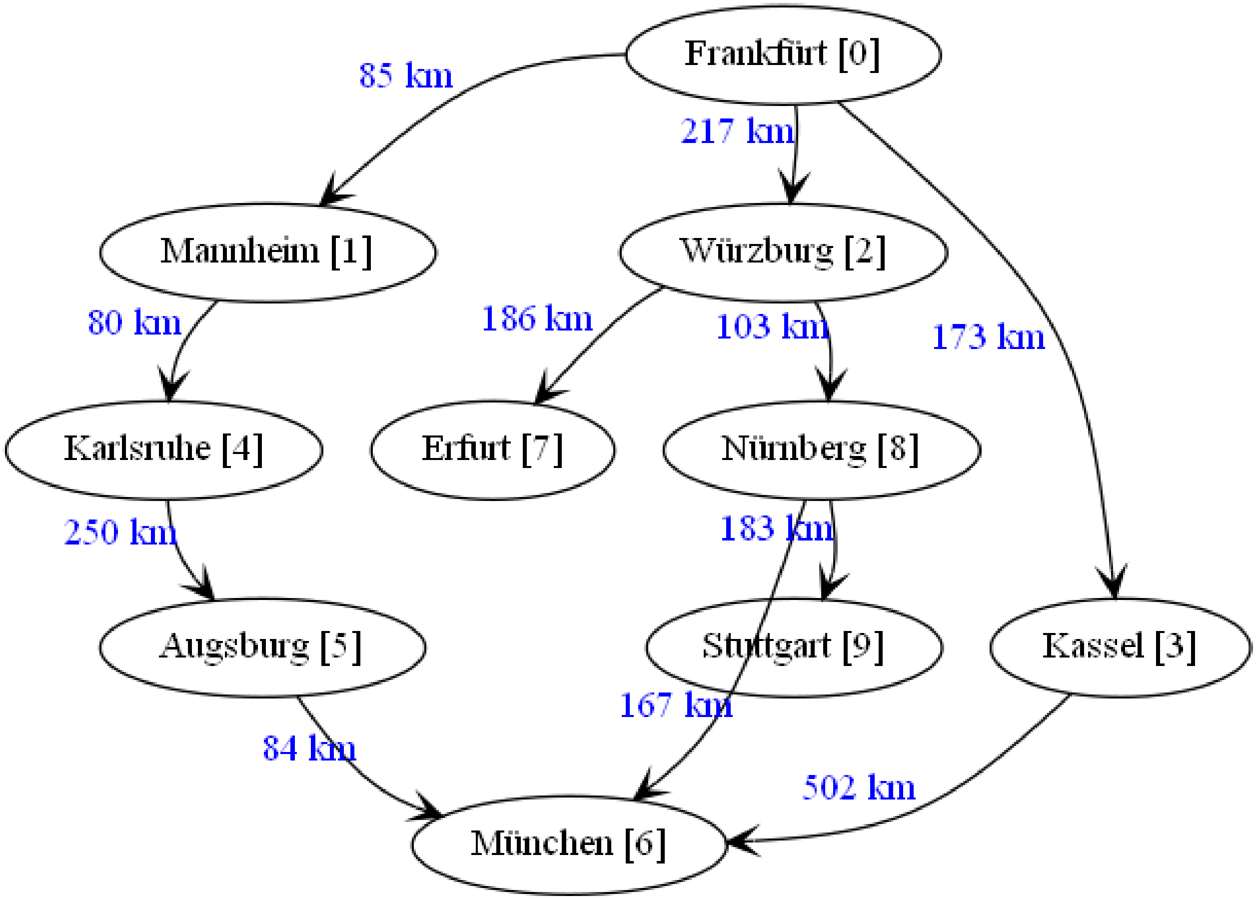
Inner Range - Edges



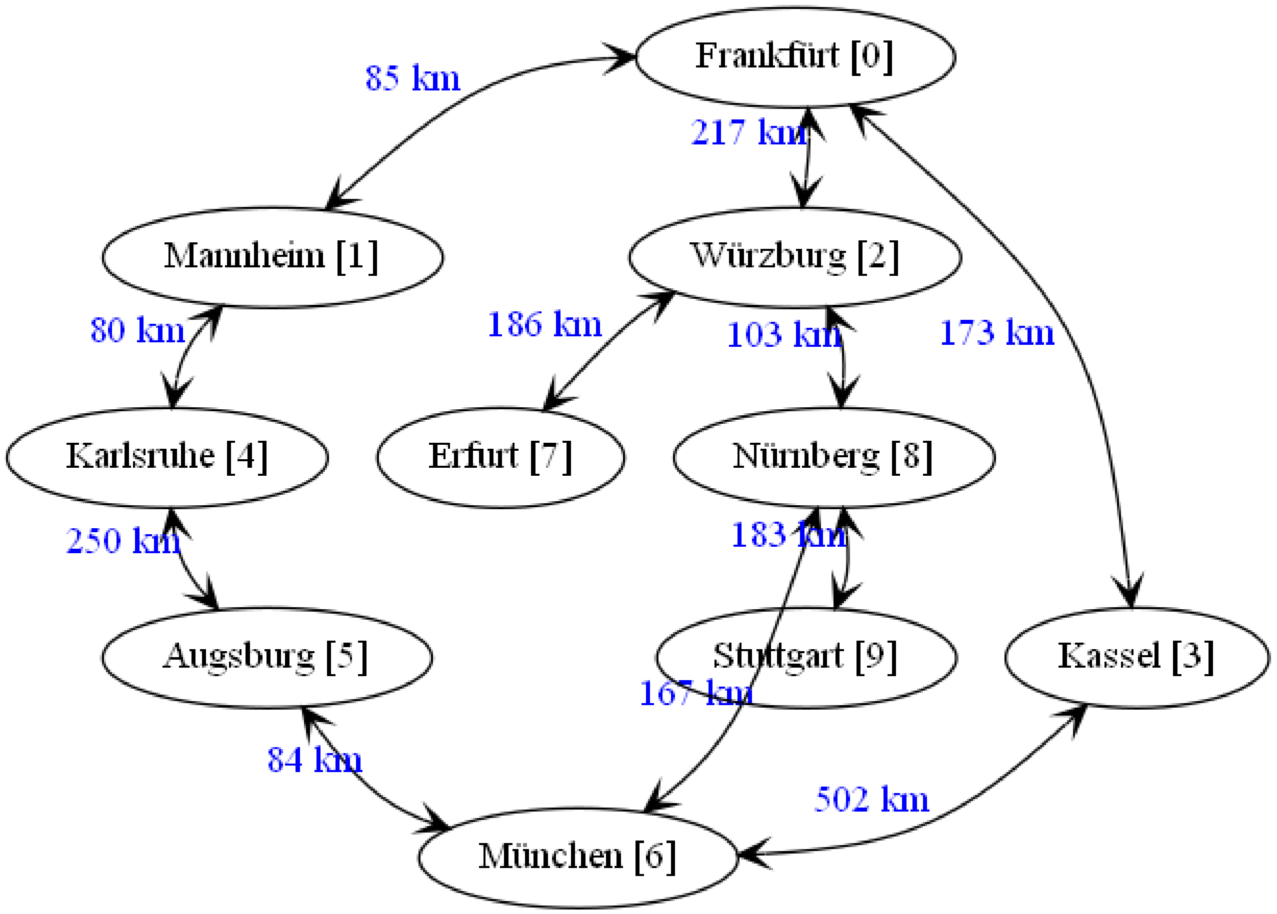

```
struct route{int target_id; double distance;}
using edges_type = std::list<route>;

struct vertex{edges_type edges; string name;}
using vertices_type = std::vector<vertex>;
```

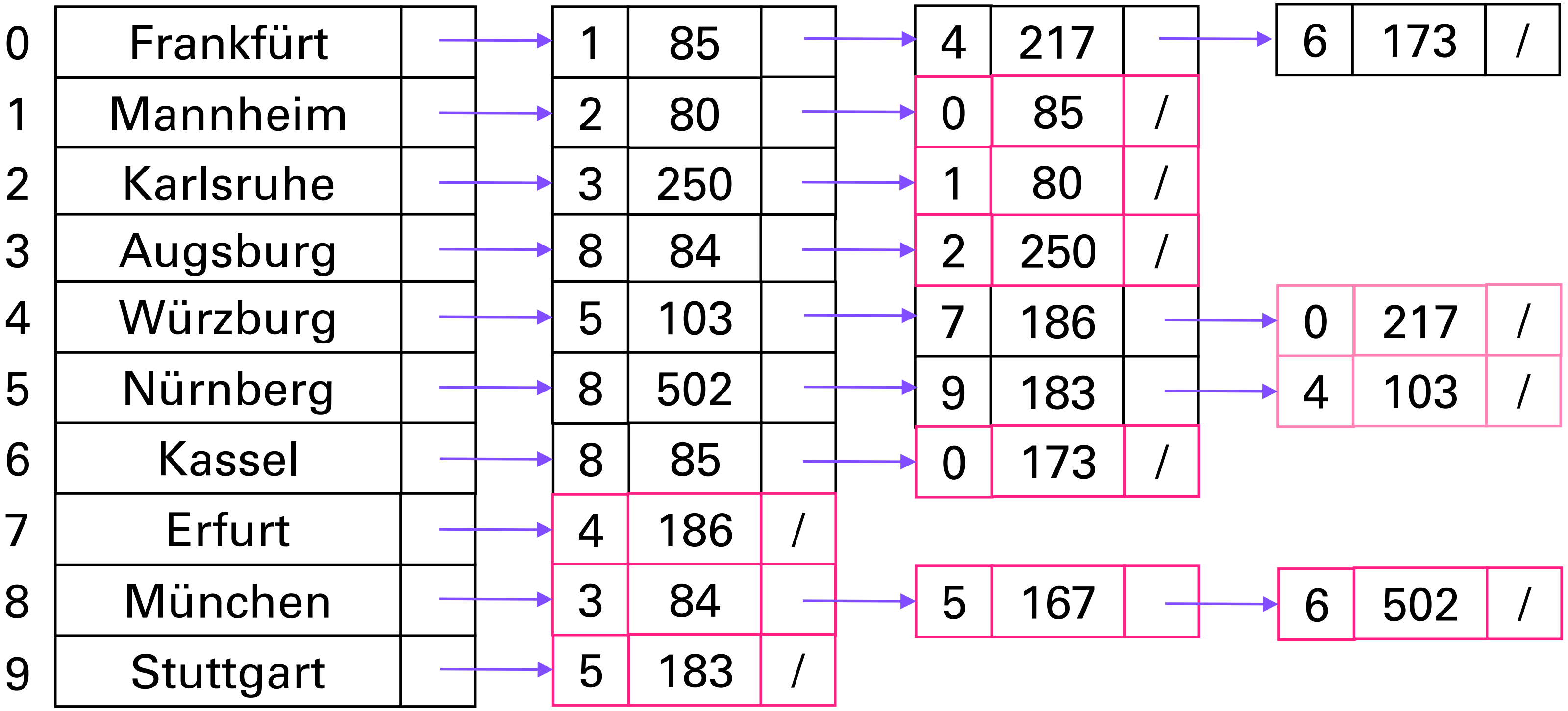
What Is a Graph?



What Is a Graph?



Adjacency List



Edge List

From	To	Distance
Frankfürt	Mannheim	85
Frankfürt	Würzburg	217
Frankfürt	Kassel	173
Mannheim	Karlsruhe	80
Karlsruhe	Augsburg	250
Augsburg	München	84
Würzburg	Erfurt	186
Würzburg	Nürnberg	103
Nürnberg	Stuttgart	183
Nürnberg	München	167
Kassel	München	502

source_id	target_id	distance
0	1	85
0	4	217
0	6	173
1	2	80
2	3	250
3	8	84
4	7	186
4	5	103
5	9	183
5	8	167
6	8	502

Other Kinds of Graphs

- Bipartite and n-partite graphs – investigation needed
- Hypergraphs – not supported

Challenges

- Enable high performance algorithms
  - What algorithms to include initially?
- How to represent a container as a range-of-ranges?
  - How to represent STL separation of container, iterator and algorithm?
  - “Bring your own graph”
- How are user-defined values defined?
- How to use modern C++ to make it easy and fun?
- Where are the boundaries?

Naming Conventions

Template Parameter	Variable Name	Description
G	g	Graph object
V	u, v, x, y	Vertex (reference)
VId	uid, vid, xid, yid, seed	Vertex Id
VR	ur, vr	Vertex Range
VI	ui, vi	Vertex Iterator
VV	val	Vertex value (user-defined type)
VVF	vvf	Vertex Value Function
E	uv, vw	Edge (reference)
ER	er	Edge Range
EI	uvi, vwi	Edge Iterator
EV	val	Edge value
EVF	evf	Edge Value Function

EXAMPLE

A Simple Graph for Routes in Germany

Graph Traversal with Views

Dijkstra’s Shortest Paths Algorithm

Raw Data – Cities & Routes

```cpp
// city data (vertices)
using city_id_type   = int32_t;
using city_name_type = string;
vector<city_name_type> city_names = {"Frankfürt", "Mannheim", "Karlsruhe", "Augsburg", "Würzburg",
                                     "Nürnberg",  "Kassel",    "Erfurt",    "München",  "Stuttgart"};

// edge data (edgelist)
using route_data = copyable_edge_t<city_id_type, double>; // {source_id, target_id, value}
vector<route_data> routes_doubled = {
    {0, 1, 85.0},  {0, 4, 217.0}, {0, 6, 173.0}, //
    {1, 0, 85.0},  {1, 2, 80.0},                 //
    {2, 1, 80.0},  {2, 3, 250.0},                //
    {3, 2, 250.0}, {3, 8, 84.0},                 //
    {4, 0, 217.0}, {4, 5, 103.0}, {4, 7, 186.0}, //
    {5, 4, 103.0}, {5, 8, 167.0}, {5, 9, 183.0}, //
    {6, 0, 173.0}, {6, 8, 502.0},                //
    {7, 4, 186.0},                               //
    {8, 3, 84.0},  {8, 5, 167.0}, {8, 6, 502.0}, //
    {9, 5, 183.0},
};
```

A Simple Graph for Routes

```
struct route { // edge
  city_id_type target_id = 0;
  double       distance  = 0.0; // km
};
using AdjList = vector<list<route>>;                  // range of ranges
using G       = rr_adaptor<AdjList, city_names_type>; // graph

G g(city_names, routes_doubled);

// Useful demo values-
city_id_type          frankfurt_id = 0;
vertex_reference_t<G> frankfurt    = *find_vertex(g, frankfurt_id);
```

Graph Traversal with Views

```
cout << "Traverse the vertices & outgoing edges" << endl;
for (auto&& [uid, u] : vertexlist(g)) {          // [id,vertex&]
  cout << city_id(g, uid) << endl;              // city name [id]

  for (auto&& [vid, uv] : incidence(g, uid)) { // [target_id,edge&]
    cout << "   --> " << city_id(g, vid) << endl;
                                  // "--> "target city" [target_id]
  }
}
```

```
Traverse the vertices & outgoing edges
Frankfürt [0]
   --> Mannheim [1]
   --> Würzburg [4]
   --> Kassel [6]
Mannheim [1]
   --> Frankfürt [0]
   --> Karlsruhe [2]
Karlsruhe [2]
   --> Mannheim [1]
   --> Augsburg [3]
Augsburg [3]
   --> Karlsruhe [2]
   --> München [8]
Würzburg [4]
   --> Frankfürt [0]
   --> Nürnberg [5]
   --> Erfurt [7]
…
München [8]
   --> Augsburg [3]
   --> Nürnberg [5]
   --> Kassel [6]
Stuttgart [9]
   --> Nürnberg [5]
```

Dijkstra’s Shortest Paths

Find the shortest paths from a seed to connected vertices

```
void dijkstra_clrs(
      G&&              g,            // graph
      vertex_id_t<G> seed,         // starting vertex_id
      Distance&        distance,     // out: distance[uid] of uid from seed
      Predecessor&     predecessor, // out: predecessor[uid] of uid in shortest path
      WF               weight = [](edge_reference_t<G> uv) { return 1; } // weight function (non-negative)
)
```

Shortest Paths - Segments

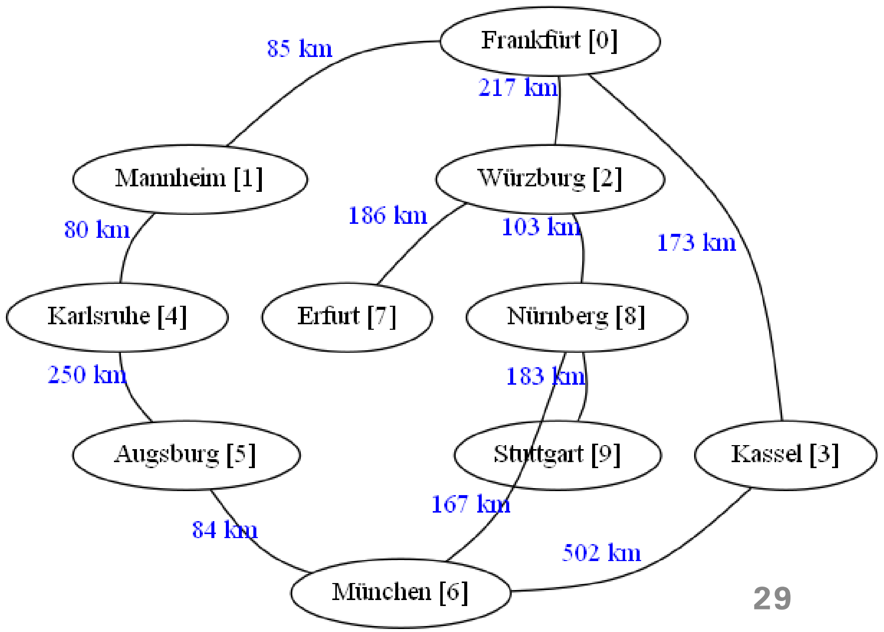
```
auto weight_1 = [](edge_reference_t<G> uv) -> int
                  { return 1; };

std::vector<int>             distance(size(vertices(g)));
std::vector<vertex_id_t<G>> predecessor(size(vertices(g)));
dijkstra_clrs(g, frankfurt_id, distance, predecessor, weight_1);

cout << "Shortest distance (segments) from "
     << city(g, frankfurt_id) << endl;

for (vertex_id_t<G> uid = 0; uid < size(vertices(g)); ++uid)
  if (distance[uid] > 0)
    cout << "  --> " << city_id(g, uid)
         << " - " << distance[uid] << " segments" << endl;
```

```
Shortest distance (segments) from Frankfürt
  --> Mannheim [1] - 1 segments
  --> Karlsruhe [2] - 2 segments
  --> Augsburg [3] - 3 segments
  --> Würzburg [4] - 1 segments
  --> Nürnberg [5] - 2 segments
  --> Kassel [6] - 1 segments
  --> Erfurt [7] - 2 segments
  --> München [8] - 2 segments
  --> Stuttgart [9] - 3 segments
```



Shortest Paths - Kilometers

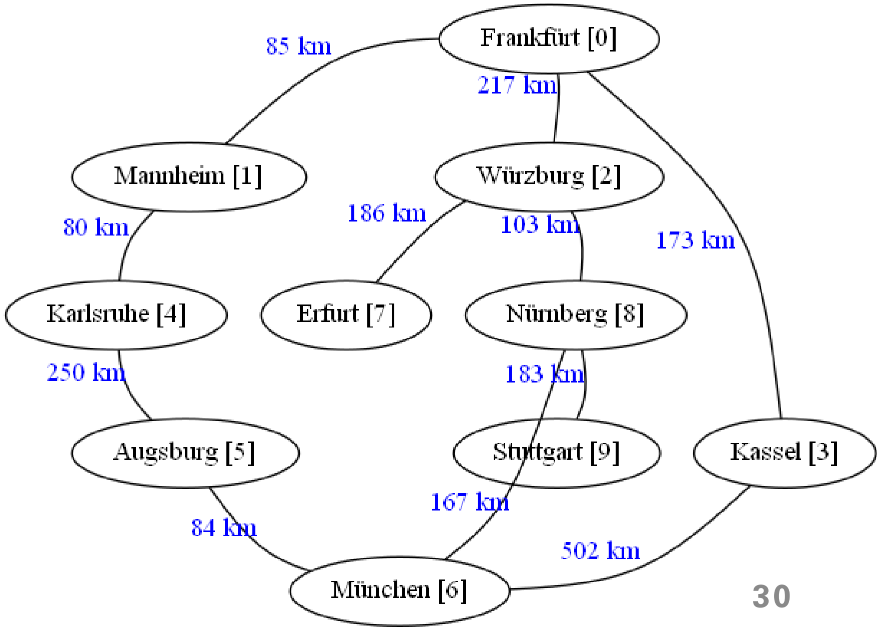
```
auto weight = [&g](edge_reference_t<G> uv)
                  { return edge_value(g, uv); }; // { return 1; };

std::vector<double>         distance(size(vertices(g)));
std::vector<vertex_id_t<G>> predecessor(size(vertices(g)));
dijkstra_clrs(g, frankfurt_id, distance, predecessor, weight);

cout << "Shortest distance (km) from "
     << city(g, frankfurt_id) << endl;

for (vertex_id_t<G> uid = 0; uid < size(vertices(g)); ++uid)
  if (distance[uid] > 0)
    cout << "  --> " << city_id(g, uid)
         << " - " << distance[uid] << "km" << endl;
```

```
Shortest distance (km) from Frankfürt
  --> Mannheim [1] - 85km
  --> Karlsruhe [2] - 165km
  --> Augsburg [3] - 415km
  --> Würzburg [4] - 217km
  --> Nürnberg [5] - 320km
  --> Kassel [6] - 173km
  --> Erfurt [7] - 403km
  --> München [8] - 487km
  --> Stuttgart [9] - 503km
```



Shortest Paths – Farthest City

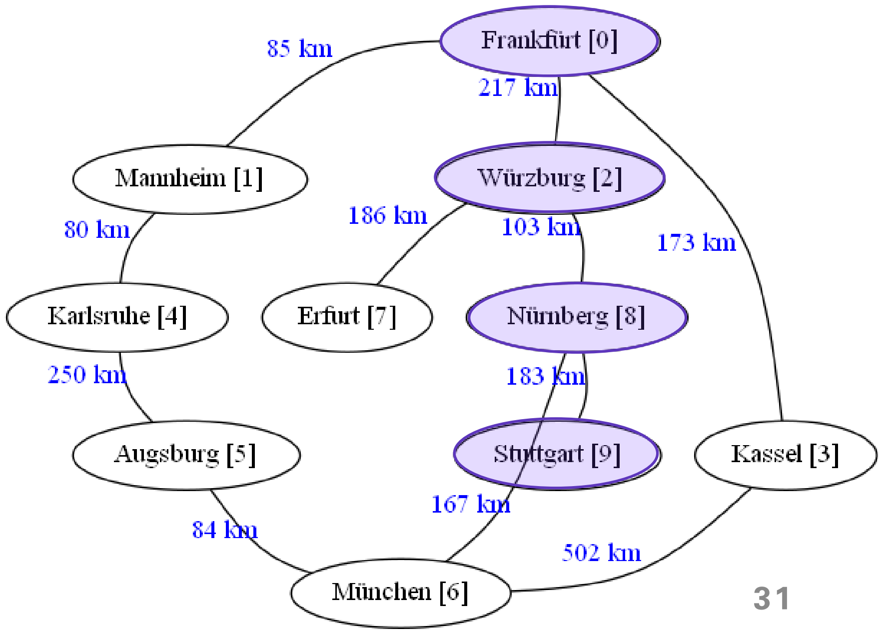
```
// Find farthest city
vertex_id_t<G> farthest_id   = frankfurt_id;
double         farthest_dist = 0.0;
for (vertex_id_t<G> uid = 0; uid < size(vertices(g)); ++uid) {
  if (distance[uid] > farthest_dist) {
    farthest_dist = distance[uid];
    farthest_id   = uid;
  }
}

cout << "The farthest city from " << city(g, frankfurt_id)
     << " is " << city(g, farthest_id)
     << " at " << distance[farthest_id] << "km" << endl;

cout << "The shortest path from " << city(g, farthest_id)
     << " to " << city(g, frankfurt_id)
     << " is: " << endl
     << "  ";

// Output path for farthest distance
for (vertex_id_t<G> uid = farthest_id; uid != frankfurt_id;
     uid = predecessor[uid]) {
  if (uid != farthest_id)
    cout << " -- ";
  cout << city_id(g, uid);
}
cout << " -- " << city_id(g, frankfurt_id) << endl;
```

```
The farthest city from Frankfürt is Stuttgart at 503km
The shortest path from Stuttgart to Frankfürt is:
  Stuttgart [9] -- Nürnberg [5] -- Würzburg [4] -- Frankfürt [0]
```



ALGORITHMS

Dijkstra Shortest Paths Interface

Concepts

Proposed Algorithms

Dijkstra Shortest Paths

```
template <adjacency_list              G,
          ranges::random_access_range Distance,
          ranges::random_access_range Predecessor,
          class WF = std::function<ranges::range_value_t<Distance>(edge_reference_t<G>)>>
requires ranges::random_access_range<vertex_range_t<G>> &&
         integral<vertex_id_t<G>> &&
         is_arithmetic_v<ranges::range_value_t<Distance>> &&
         convertible_to<vertex_id_t<G>, ranges::range_value_t<Predecessor>> &&
         edge_weight_function<G, WF>
void dijkstra_clrs(
      G&&            g,            // graph
      vertex_id_t<G> seed,         // starting vertex_id
      Distance&      distance,     // out: distance[uid] = distance of uid from seed
      Predecessor&   predecessor,  // out: predecessor[uid] = prev id of uid in shortest path
      WF             weight = [](edge_reference_t<G> uv) // default: weight(uv) -> 1
                          { return ranges::range_value_t<Distance>(1); }
);
```

Dijkstra Shortest Paths

```
template <adjacency_list              G,
         ranges::random_access_range Distance,
         ranges::random_access_range Predecessor,
         class WF = std::function<ranges::range_value_t<Distance>(edge_reference_t<G>)>>
requires ranges::random_access_range<vertex_range_t<G>> &&
         integral<vertex_id_t<G>> &&
         is_arithmetic_v<ranges::range_value_t<Distance>> &&
         convertible_to<vertex_id_t<G>, ranges::range_value_t<Predecessor>> &&
         edge_weight_function<G, WF>
void dijkstra_clrs(
      G&&            g,            // graph
      vertex_id_t<G> seed,         // starting vertex_id
      Distance&      distance,     // out: distance[uid] = distance of uid from seed
      Predecessor&   predecessor,  // out: predecessor[uid] = prev id of uid in shortest path
      WF             weight = [](edge_reference_t<G> uv) // default: weight(uv) -> 1
                         { return ranges::range_value_t<Distance>(1); }
);
```

Dijkstra Shortest Paths

```
template <adjacency_list             G,
          ranges::random_access_range Distance,
          ranges::random_access_range Predecessor,
          class WF = std::function<ranges::range_value_t<Distance>(edge_reference_t<G>)>>
requires ranges::random_access_range<vertex_range_t<G>> &&
         integral<vertex_id_t<G>> &&
         is_arithmetic_v<ranges::range_value_t<Distance>> &&
         convertible_to<vertex_id_t<G>, ranges::range_value_t<Predecessor>> &&
         edge_weight_function<G, WF>
void dijkstra_clrs(
      G&&             g,           // graph
      vertex_id_t<G>  seed,        // starting vertex_id
      Distance&       distance,    // out: distance[uid] = distance of uid from seed
      Predecessor&    predecessor, // out: predecessor[uid] = prev id of uid in shortest path
      WF              weight = [](edge_reference_t<G> uv) // default: weight(uv) -> 1
                           { return ranges::range_value_t<Distance>(1); }
);
```

Dijkstra Shortest Paths

```
template <adjacency_list                G,
          ranges::random_access_range Distance,
          ranges::random_access_range Predecessor,
          class WF = std::function<ranges::range_value_t<Distance>(edge_reference_t<G>)>>
requires ranges::random_access_range<vertex_range_t<G>> &&
         integral<vertex_id_t<G>> &&
         is_arithmetic_v<ranges::range_value_t<Distance>> &&
         convertible_to<vertex_id_t<G>, ranges::range_value_t<Predecessor>> &&
         edge_weight_function<G, WF>
void dijkstra_clrs(
      G&&             g,            // graph
      vertex_id_t<G>  seed,         // starting vertex_id
      Distance&       distance,     // out: distance[uid] = distance of uid from seed
      Predecessor&    predecessor,  // out: predecessor[uid] = prev id of uid in shortest path
      WF              weight = [](edge_reference_t<G> uv) // default: weight(uv) -> 1
                                 { return ranges::range_value_t<Distance>(1); }
);
```

Dijkstra Shortest Paths

```
template <adjacency_list                G,
          ranges::random_access_range Distance,
          ranges::random_access_range Predecessor,
          class WF = std::function<ranges::range_value_t<Distance>(edge_reference_t<G>)>>
requires ranges::random_access_range<vertex_range_t<G>> &&
         integral<vertex_id_t<G>> &&
         is_arithmetic_v<ranges::range_value_t<Distance>> &&
         convertible_to<vertex_id_t<G>, ranges::range_value_t<Predecessor>> &&
         edge_weight_function<G, WF>
void dijkstra_clrs(
      G&&            g,            // graph
      vertex_id_t<G> seed,         // starting vertex_id
      Distance&      distance,     // out: distance[uid] = distance of uid from seed
      Predecessor&   predecessor,  // out: predecessor[uid] = prev id of uid in shortest path
      WF             weight = [](edge_reference_t<G> uv) // default: weight(uv) -> 1
                         { return ranges::range_value_t<Distance>(1); }
);
```

edge_weight_function Concept

```
template <class G, class F>
concept edge_weight_function = // e.g. weight(uv)
      is_arithmetic_v<invoke_result_t<WF, edge_reference_t<G>>>;
```

vertex_range Concept

```
template <class G>
using vertex_range_t = decltype(vertices(declval<G&&>()));
template <class G>
using vertex_iterator_t = ranges::iterator_t<vertex_range_t<G&&>>;
template <class G>
using vertex_reference_t = ranges::range_reference_t<vertex_range_t<G>>;

template <class G>
concept vertex_range = ranges::forward_range<vertex_range_t<G>> &&
                       ranges::sized_range<vertex_range_t<G>> &&
requires(G&& g, vertex_iterator_t<G> ui) {
  { vertices(g) } -> ranges::forward_range;
  vertex_id(g, ui);
};
```

targeted_edge Concept

```
template <class G>
using vertex_edge_range_t
      = decltype(edges(declval<G&&>(), declval<vertex_reference_t<G>>()));

template <class G>
using edge_reference_t = ranges::range_reference_t<vertex_edge_range_t<G>>;

template <class G>
concept targeted_edge = requires(G&& g, edge_reference_t<G> uv) {
  target_id(g, uv);
  target(g, uv);
};
```

adjacency_list Concept

```
template <class G>
concept adjacency_list =
    vertex_range<G> &&
    targeted_edge<G, edge_t<G>> &&
    requires( G&&                  g,
              vertex_reference_t<G> u,
              vertex_id_t<G>        uid) {
      { edges(g, u) } -> ranges::forward_range;
      { edges(g, uid) } -> ranges::forward_range;
    };
```

sourced_adjacency_list Concept

```
template <class G, class E>
concept sourced_edge =
  requires(G&& g, E& uv) {
    source_id(g, uv);
    source(g, uv);
  };

template <class G>
concept sourced_adjacency_list =
  adjacency_list<G> &&
  sourced_edge<G, edge_t<G>> &&
  requires(G&& g, edge_reference_t<G> uv) {
    edge_id(g, uv);
  };
```

Proposed Algorithms

Confirmed

- Dijkstra Shortest Path
- Bellman-Ford Shortest Path
- Connected Components
- Strongly Connected Components
- Biconnected Components
- Articulation Points
- Minimum Spanning Tree

Candidates

- Page Rank
- Betweenness Centrality
- Triangle Count
- Subgraph Isomorphism
- Kruskal Minimum Spanning Tree
- Prim Minimum Spanning Tree
- Louvain (Community Detection)
- Label Propagation (Community Detection)

VIEWS

vertexlist and vertex_view

incidence and edge_view

neighbors and neighbor_view

edgelist

depth_first_search

breadth_first_search

topological_sort

vertexlist(g,u) and vertex_view

```
G g = …;
for (auto&& uu : vertexlist(g)) {
  vertex_id_t<G>         uid = uu.id;
  vertex_reference_t<G> u    = uu.vertex;
  // do something interesting
}
```

```
template <class VId, class V>
struct vertex_view<VId, V, void> {
  VId id;
  V   vertex;
};
```

vertexlist(g,u) and vertex_view

```
G g = …;
for (auto&& uu : vertexlist(g)) {
  vertex_id_t<G>         uid = uu.id;
  vertex_reference_t<G> u   = uu.vertex;
  // do something interesting
}
```

```
for (auto&& [uid,u] : vertexlist(g)) {
  // do something interesting
}
```

```
template <class VId, class V>
struct vertex_view<VId, V, void> {
  VId id;
  V   vertex;
};
```

vertexlist(g,u,vvf) and vertex_view

```
G g = …;
for (auto&& uu : vertexlist(g)) {
  vertex_id_t<G>         uid = uu.id;
  vertex_reference_t<G> u    = uu.vertex;
  // do something interesting
}
```

```
for (auto&& [uid,u] : vertexlist(g)) {
  // do something interesting
}
```

```
auto&& vvf = [&g](vertex_reference_t<G> u)
                 { return vertex_value(g, u); };
for (auto&& [uid, u, val] : vertexlist(g,vvf)) {
  // do something interesting
}
```

```
template <class VId, class V, class VV>
struct vertex_view {
  VId id;
  V   vertex;
  VV  value;
};
```

```
template <class VId, class V>
struct vertex_view<VId, V, void> {
  VId id;
  V   vertex;
};
```

vertexlist(g,u[,vvf]) and vertex_view

```
G g = …;
for (auto&& uu : vertexlist(g)) {
  vertex_id_t<G>        uid = uu.id;
  vertex_reference_t<G> u   = uu.vertex;
  // do something interesting
}
```

```
for (auto&& [uid,u] : vertexlist(g)) {
  // do something interesting
}
```

```
auto&& vvf = [&g](vertex_reference_t<G> u)
                 { return vertex_value(g, u); };
for (auto&& [uid, u, val] : vertexlist(g,vvf)) {
  // do something interesting
}
```

```
template <class VId, class V, class VV>
struct vertex_view {
  VId id;
  V   vertex;
  VV  value;
};
```

```
template <class VId, class V>
struct vertex_view<VId, V, void> {
  VId id;
  V   vertex;
};
```

```
template <class VId, class VV>
struct vertex_view<VId, void, VV> {
  VId id;
  VV  value;
};
```

```
template <class VId>
struct vertex_view<VId, void, void> {
  VId id;
};
```

vertexlist overloads

Example	Return Type
for(auto&& [uid,u] : vertexlist(g))	vertex_view<VId,V,void>
for(auto&& [uid,u,val] : vertexlist(g,vvf))	vertex_view<VId,V,VV>
for(auto&& [uid,u] : vertexlist(g,first,last))	vertex_view<VId,V,void>
for(auto&& [uid,u,val] : vertexlist(g,first,last,vvf))	vertex_view<VId,V,VV>
for(auto&& [uid,u] : vertexlist(g,vr))	vertex_view<VId,V,void>
for(auto&& [uid,u,val] : vertexlist(g,vr,vvf))	vertex_view<VId,V,VV>

incidence(g,u) and edge_view

```
G g = …;
vertex_reference_t<G> u = ...;
for (auto&& [vid, uv] : incidence(g, uid)) {
  // do something interesting...
}
```

```
template <class VId, class E>
struct edge_view<VId, false, E, void> {
  VId target_id;
  E   edge;
};
```

incidence(g,u,evf) and edge_view

```
G g = …;
vertex_reference_t<G> u = ...;
for (auto&& [vid, uv] : incidence(g, uid)) {
  // do something interesting...
}
```

```
template <class VId, class E>
struct edge_view<VId, false, E, void> {
  VId target_id;
  E   edge;
};
```

```
auto edge_fn = [&g](edge_reference_t<G> uv)
                   { return edge_value(g, uv); };
for (auto&& [vid, uv, val] : incidence(g, uid, edge_fn))
{
  // do something interesting...
}
```

```
template <class VId, class E , class EV>
struct edge_view<VId, false, E, EV> {
  VId target_id;
  E   edge;
  VV  value;
};
```

incidence(g,u[,evf]) and edge_view

```
template <class VId, bool Sourced,
          class E, class EV>
struct edge_view {
  VId source_id;
  VId target_id;
  E   edge;
  EV  value;
};
```

```
G g = …;
vertex_reference_t<G> u = ...;
for (auto&& [vid, uv] : incidence(g, uid)) {
  // do something interesting...
}
```

```
template <class VId, class E>
struct edge_view<VId, false, E, void> {
  VId target_id;
  E   edge;
};
```

```
auto edge_fn = [&g](edge_reference_t<G> uv)
                   { return edge_value(g, uv); };
for (auto&& [vid, uv, val] : incidence(g, uid, edge_fn))
{
  // do something interesting...
}
```

```
template <class VId, class E , class EV>
struct edge_view<VId, false, E, EV> {
  VId target_id;
  E   edge;
  VV  value;
};
```

edge_view<Vid,Sourced,E,EV>

Template Arguments	Member Variables			
edge_view<VId, true, E, EV>	source_id	target_id	edge	value
edge_view<VId, true, E, void>	source_id	target_id	edge	
edge_view<VId, true, void, EV>	source_id	target_id		value
edge_view<VId, true, void, void>	source_id	target_id		
edge_view<VId, false, E, EV>		target_id	edge	value
edge_view<VId, false, E, void>		target_id	edge	
edge_view<VId, false, void, EV>		target_id		value
edge_view<VId, false, void, void>		target_id		

neighbors(g,u) and neighbor_view

```cpp
G g = …;
for (auto&& [vid, v] : neighbors(g, uid)) {
  // do something interesting...
}
```

```cpp
template <class VId, class V>
struct neighbor_view<VId, false, V, void> {
  VId target_id;
  V   target;
};
```

neighbors(g,u,evf) and neighbor_view

```
G g = …;
for (auto&& [vid, v] : neighbors(g, uid)) {
  // do something interesting...
}
```

```
template <class VId, class V>
struct neighbor_view<VId, false, V, void> {
  VId target_id;
  V   target;
};
```

```
auto vvf = [&g](vertex_reference_t<G> v)
               { return vertex_value(g, v); };
for (auto&& [vid, v, val] : neighbors(g, uid, vvf))
{
  // do something interesting...
}
```

```
template <class VId, class V, class VV>
struct neighbor_view<VId, false, V, VV> {
  VId target_id;
  V   target;
  VV  value;
};
```

neighbors(g,u[,evf]) and neighbor_view

```
template <class VId, bool Sourced,
          class V, class VV>
struct neighbor_view {
  VId source_id;
  VId target_id;
  V   target;
  VV  value;
};
```

```
G g = …;
for (auto&& [vid, v] : neighbors(g, uid)) {
  // do something interesting...
}
```

```
template <class VId, class V>
struct neighbor_view<VId, false, V, void> {
  VId target_id;
  V   target;
};
```

```
auto vvf = [&g](vertex_reference_t<G> v)
                { return vertex_value(g, v); };
for (auto&& [vid, v, val] : neighbors(g, uid, vvf))
{
  // do something interesting...
}
```

```
template <class VId, class V, class VV>
struct neighbor_view<VId, false, V, VV> {
  VId target_id;
  V   target;
  VV  value;
};
```

neighbor_view<Vid,Sourced,V,VV>

Template Arguments	Member Variables			
neighbor_view<VId, true, E, EV>	source_id	target_id	target	value
neighbor_view<VId, true, E, void>	source_id	target_id	target	
neighbor_view<VId, true, void, EV>	source_id	target_id		value
neighbor_view<VId, true, void, void>	source_id	target_id		
neighbor_view<VId, false, E, EV>		target_id	target	value
neighbor_view<VId, false, E, void>		target_id	target	
neighbor_view<VId, false, void, EV>		target_id		value
neighbor_view<VId, false, void, void>		target_id		

edgelist(g,u)

```
G g = …;
for (auto&& [uid, vid, uv] : edgelist(g)) {
  // do something interesting...
}
```

```
template <class VId, class E>
struct edge_view<VId, true, E, void> {
  VId source_id;
  VId target_id;
  E   edge;
};
```

edgelist(g,u,evf)

```cpp
auto evf = [&g](edge_reference_t<G2> uv)
               { return edge_value(g, uv); };
for (auto&& [uid, vid, uv, val] : edgelist(g, evf)) {
  // do something interesting...
}
```

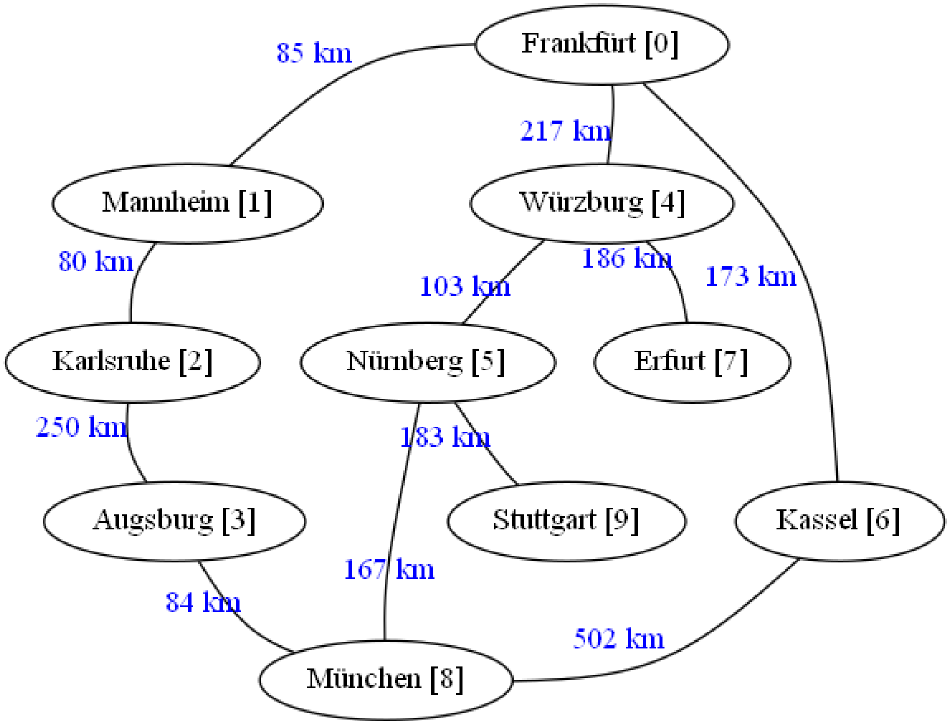
```cpp
template <class VId, bool Sourced,
          class E, class EV>
struct edge_view {
  VId source_id;
  VId target_id;
  E   edge;
  EV  value;
};
```

```cpp
G g = …;
for (auto&& [uid, vid, uv] : edgelist(g)) {
  // do something interesting...
}
```

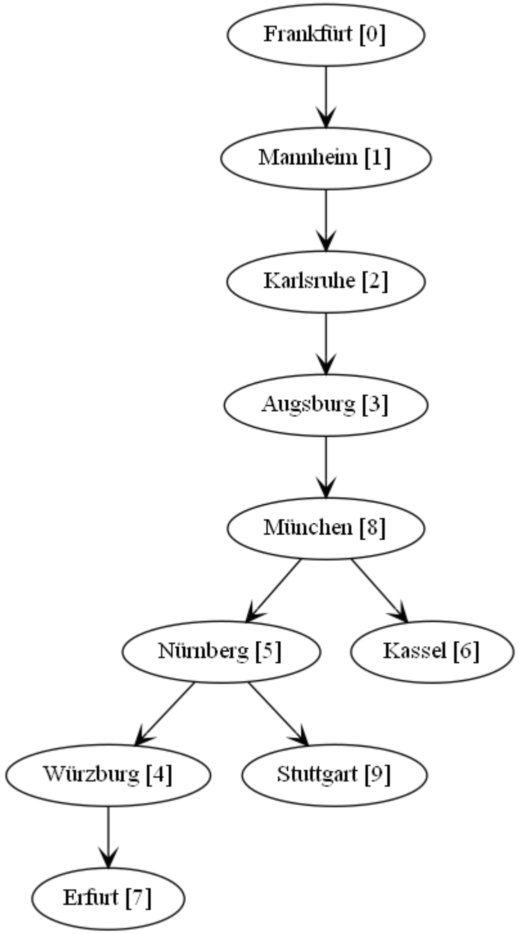
```cpp
template <class VId, class E>
struct edge_view<VId, true, E, void> {
  VId source_id;
  VId target_id;
  E   edge;
};
```

Depth First Search

routes



Depth First Search

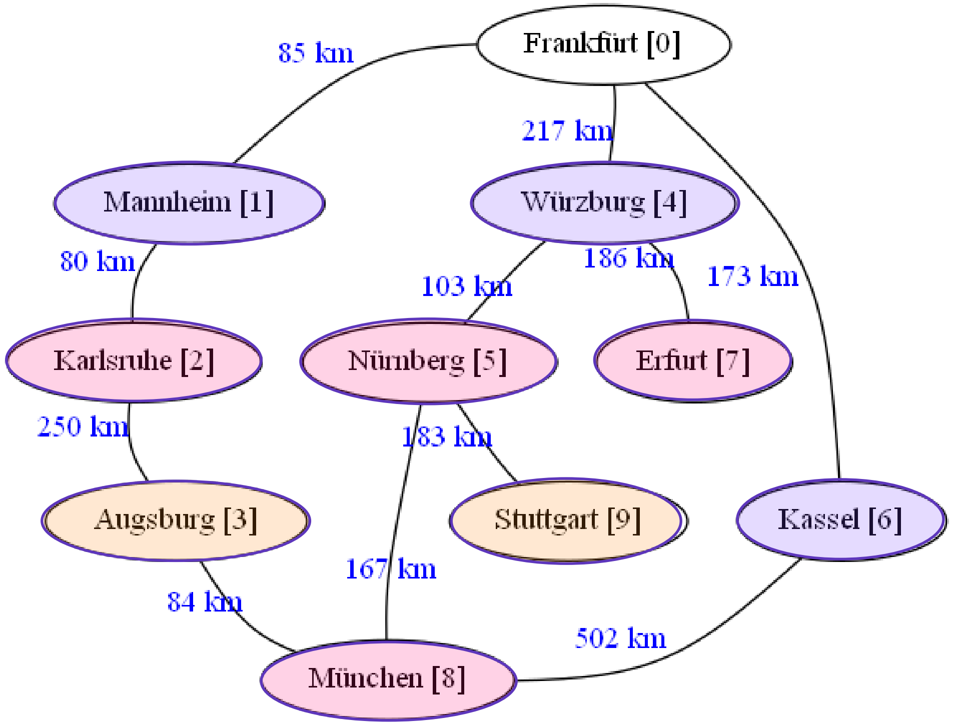


depth_first_search functions

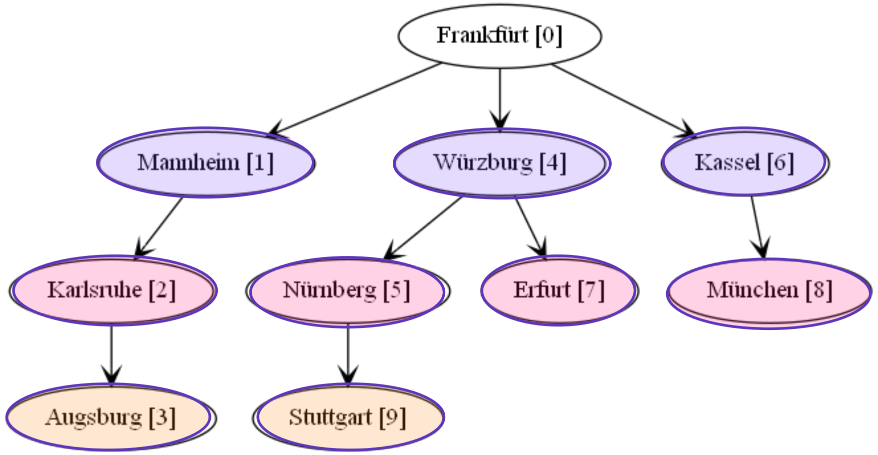
Example	Return Type
for(auto&& [vid,v] : vertices_depth_first_search(g,seed))	vertex_view<VId,V,void>
for(auto&& [vid,v,val] : vertices_depth_first_search(g,seed,vvf))	vertex_view<VId,V,VV>
for(auto&& [vid,uv] : edges_depth_first_search(g,seed))	edge_view<VId,false,E,void>
for(auto&& [vid,uv,val] : edges_depth_first_search(g,seed,evf))	edge_view<VId,false,E,EV>
for(auto&& [uid,vid,uv] : sourced_edges_depth_first_search(g,seed))	edge_view<VId,true,E,void>
for(auto&& [uid,vid,uv,val] : sourced_edges_depth_first_search(g,seed,evf))	edge_view<VId,true,E,EV>

Breadth First Search

routes



Breadth First Search

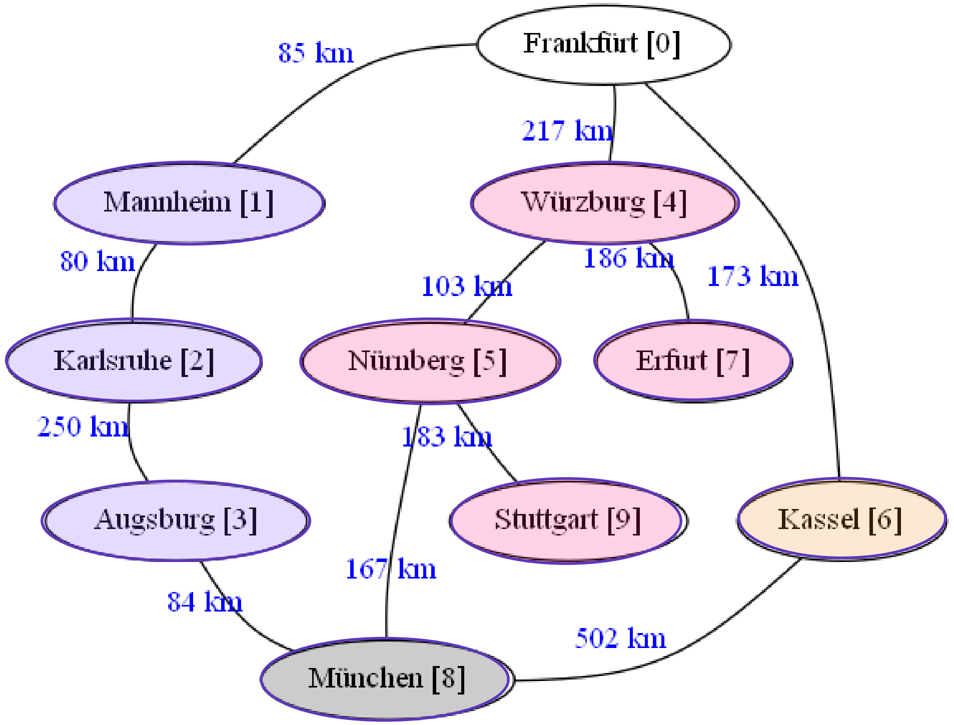


breadth_first_search functions

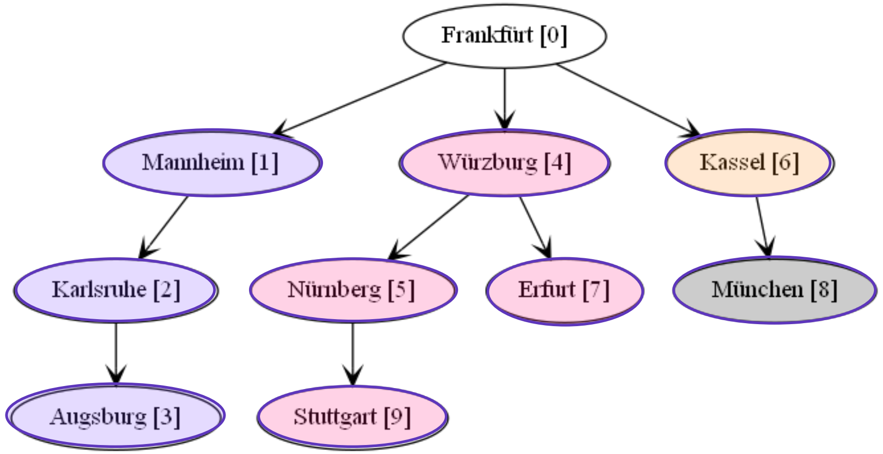
Example	Return Type
for(auto&& [vid,v] : vertices_breadth_first_search(g,seed))	vertex_view<VId,V,void>
for(auto&& [vid,v,val] : vertices_breadth_first_search(g,seed,vvf))	vertex_view<VId,V,VV>
for(auto&& [vid,uv] : edges_breadth_first_search(g,seed))	edge_view<VId,false,E,void>
for(auto&& [vid,uv,val] : edges_breadth_first_search(g,seed,evf))	edge_view<VId,false,E,EV>
for(auto&& [uid,vid,uv] : sourced_edges_breadth_first_search(g,seed))	edge_view<VId,true,E,void>
for(auto&& [uid,vid,uv,val] : sourced_edges_breadth_first_search(g,seed,evf))	edge_view<VId,true,E,EV>

Topological Sort

routes



Topological Sort



topological_sort functions

Example	Return Type
for(auto&& [vid,v] : vertices_topological_sort(g,seed))	vertex_view<VId,V,void>
for(auto&& [vid,v,val] : vertices_topological_sort (g,seed,vvf))	vertex_view<VId,V,VV>
for(auto&& [vid,uv] : edges_topological_sort(g,seed))	edge_view<VId,false,E,void>
for(auto&& [vid,uv,val] : edges_topological_sort(g,seed,evf))	edge_view<VId,false,E,EV>
for(auto&& [uid,vid,uv] : sourced_edges_topological_sort(g,seed))	edge_view<VId,true,E,void>
for(auto&& [uid,vid,uv,val] : sourced_edges_topological_sort(g,seed,evf))	edge_view<VId,true,E,EV>

CSR_GRAPH GRAPH CONTAINER

csr_graph

Graph Container Interface

Graph Containers: Unique Among Containers

- Range of ranges
- All functions are free functions (c.f. begin, end, size, empty)
- All functions are customization points
- User-defined values are optional on edge, vertex and graph

csr_graph Graph Container

- Compressed Sparse Row (Matrix)
- High performance
- Compact memory use
- Static structure – can’t change after construction
- Values can change
- Values are stored separately from structure

csr_graph

```
template <class EV      = void,      // edge value type
          class VV      = void,      // vertex value type
          class GV      = void,      // graph value type
          integral VId = uint32_t, // vertex id type
          class Alloc  = allocator<uint32_t>> // for internal containers
class csr_graph;
```

```
using G = std::graph::csr_graph<double, std::string_view, std::string>;
```

csr_graph Edge-only Constructor

```
template <ranges::forward_range ERng, class EProj = identity>
requires copyable_edge<invoke_result<EProj, ranges::range_value_t<ERng>>, VId, EV>
constexpr csr_graph(const ERng& erng,
                    EProj eproj = {}, // eproj(eval) -> edge_view{source_id,target_id [,value]}
                    const Alloc& alloc = Alloc());
```

csr_graph Edge-only Constructor

```
template <ranges::forward_range ERng, class EProj = identity>
requires copyable_edge<invoke_result<EProj, ranges::range_value_t<ERng>>, VId, EV>
constexpr csr_graph(const ERng& erng,
                    EProj eproj = {}, // eproj(eval) -> edge_view{source_id,target_id [,value]}
                    const Alloc& alloc = Alloc());
```

csr_graph Edge and Vertex Constructor

```
template <ranges::forward_range ERng, class EProj = identity,
          ranges::forward_range VRng, class VProj = identity>
  requires copyable_edge<invoke_result<EProj, ranges::range_value_t<ERng>>, VId, EV> &&
           copyable_vertex<invoke_result<VProj, ranges::range_value_t<VRng>>, VId, VV>
  constexpr csr_graph(const ERng&  erng,
                      const VRng&  vrng,
                      EProj        eproj = {}, // eproj(eval) -> edge_view{source_id,target_id [,value]}
                      VProj        vproj = {}, // vproj(vval) -> vertex_view{id [,value]}
                      const Alloc& alloc = Alloc());
```

Graph Container Interface

- Functions
- Types
- Traits
- Concepts

Graph and Vertex Functions

Function	Return Type	Complex	Default
graph_value(g)	graph_value_t<G>	Constant	n/a, optional
vertices(g)	vetex_range_t<G>	Constant	n/a
vertex_id(g,ui)	vertex_id_t<G>	Constant	ui - begin(vertices(g))
vertex_value(g,u)	vertex_value_t<G>	Constant	n/a, optional
degree(g,u)	integral	Constant	size(edges(g,u))
find_vertex(g,uid)	vertex_iterator_t<G>	Constant	begin(vertices(g)) + uid

Edge Functions

Function	Return Type	Complex	Default
edges(g,u)	vertex_edge_range_t<G>	Constant	n/a
edges(g,uid)	vertex_edge_range_t<G>	Constant	edges(g,*find_vertex(g,uid))
target_id(g,uv)	vertex_id_t<G>	Constant	n/a
target(g,uv)	vertex_t<G>		*(begin(vertices(g)) + target_id(g, uv))
edge_value(g,uv)	edge_value_t<G>	Constant	n/a, optional
find_vertex_edge(g,u,vid)	vertex_edge_t<G>	Linear	find(edges(g,u), [](uv) {target_id(g,uv)==vid;\})}
find_vertex_edge(g,uid,vid)	vertex_edge_t<G>	Linear	find_vertex_edge(g,*find_vertex (g,uid),vid)
contains_edge(g,uid,vid)	bool	Linear	find_vertex_edge(g,uid) != end(edges(g,uid))

Sourced Edge Functions

Function	Return Type	Complex	Default
source_id(g,uv)	vertex_id_t<G>	Constant	n/a, optional
source(g,uv)	vertex_t<G>	Constant	*(begin(vertices(g)) + source_id(g,uv))
edge_id(g,uv)	edge_id_t<G>	Constant	pair(source_id(g,uv),target_id(g,uv))

Graph and Vertex Types

Type Alias	Definition	Comment
graph_reference_t<G>	add_lvalue_reference<G>	
graph_value_t<G>	decltype(graph_value(g))	Optional
vertex_range_t<G>	decltype(vertices(g))	
vertex_iterator_t<G>	iterator_t<vertex_range_t<G>>	
vertex_t<G>	range_value_t<vertex_range_t<G>>	
vertex_reference_t<G>	range_reference_t<vertex_range_t<G>>	
vertex_id_t<G>	decltype(vertex_id(g))	
vertex_value_t<G>	decltype(vertex_value(g))	Optional

Edge Types

Type Alias	Definition	Comment
vertex_edge_range_t<G>	decltype(edges(g,u))	
vertex_edge_iterator_t<G>	iterator_t<vertex_edge_range_t<G>>	
edge_t<G>	range_value_t<vertex_edge_range_t<G>>	
edge_reference_t<G>	range_reference_t<vertex_edge_range_t<G>>	
edge_value_t<G>	decltype(edge_value(g))	Optional
edge_id_t<G>	decltype(pair(source_id(g,uv),target_id(g,uv)))	sourced_edge<G>

Traits

```
template <class G>
concept has_degree = requires(G&& g, vertex_reference_t<G> u) {
  { degree(g, u) };
};

template <class G>
concept has_find_vertex = requires(G&& g, vertex_id_t<G> uid) {
  { find_vertex(g, uid) } -> forward_iterator;
};

template <class G>
concept has_find_vertex_edge = requires(G&& g, vertex_id_t<G> uid, vertex_id_t<G> vid, vertex_reference_t<G> u)
{
  { find_vertex_edge(g, u, vid) } -> forward_iterator;
  { find_vertex_edge(g, uid, vid) } -> forward_iterator;
};

template <class G>
concept has_contains_edge = requires(G&& g, vertex_id_t<G> uid, vertex_id_t<G> vid) {
  { contains_edge(g, uid, vid) } -> convertible_to<bool>;
};
```

unordered_edge and ordered_edge Traits

```
// specialized for graph container’s edge type
template <class E>
struct define_unordered_edge : public false_type {};

template <class G, class E>
struct is_unordered_edge : public conjunction<define_unordered_edge<E>, is_sourced_edge<G, E>> {};

template <class G, class E>
inline constexpr bool is_unordered_edge_v = is_unordered_edge<G, E>::value;

template <class G, class E>
concept unordered_edge = is_unordered_edge_v<G, E>;


template <class G, class E>
struct is_ordered_edge : public negation<is_unordered_edge<G,E>> {};

template <class G, class E>
inline constexpr bool is_ordered_edge_v = is_ordered_edge<G, E>::value;

template <class G, class E>
concept ordered_edge = is_ordered_edge_v<G, E>;
```

adjacency_matrix Trait

```cpp
// specialized for graph container
template <class G>
struct define_adjacency_matrix : public false_type {};

template <class G>
struct is_adjacency_matrix : public define_adjacency_matrix<G> {};

template <class G>
inline constexpr bool is_adjacency_matrix_v = is_adjacency_matrix<G>::value;

template <class G>
concept adjacency_matrix = is_adjacency_matrix_v<G>;
```

Concepts

OTHER GRAPH CONTAINERS

Integrating External Graphs

csr_partite_graph?

rr_adaptor

dynamic_graph

Integrating External Graphs

Required Function Overloads

- `vertices(g)`
- `edges(g,u)`
- `target_id(g,uv)`

Optional Function Overloads

- Value functions
  - `graph_value(g)`
  - `vertex_value(g,u)`
  - `edge_value(g,uv)`
- `vertex_id(g,ui)` – defines `vertex_id_t<G>`
- `source_id(g,uv)`

Integrating External Graphs – niebloid

```
template <range_of_ranges Outer, std::ranges::random_access_range VVR>
requires std::ranges::contiguous_range<Outer> &&
         std::ranges::random_access_range<VVR>
class rr_adaptor {
public:
  using vertices_range = Outer;

 constexpr vertices_range vertices() noexcept { return vertices_; }
 constexpr vertices_range vertices() const noexcept { return vertices_; }
private:
  vertices_range vertices_;
}
```

```
template <range_of_ranges Outer, std::ranges::random_access_range VVR>
constexpr auto vertices(rr_adaptor<Outer,VVR>& g) {
  return g.vertices();
}
template <range_of_ranges Outer, std::ranges::random_access_range VVR>
constexpr auto vertices(const rr_adaptor<Outer,VVR>& g) {
  return g.vertices();
}
```

Integrating External Graphs – tag_invoke

```
template <range_of_ranges Outer, std::ranges::random_access_range VVR>
requires std::ranges::contiguous_range<Outer> &&
         std::ranges::random_access_range<VVR>
class rr_adaptor {
public:
  using vertices_range = Outer;

private:
  friend constexpr vertices_range&
  tag_invoke(std::tag_invoke::vertices_fn_t, graph_type& g) {
      return g.vertices_;
  }
  friend constexpr const vertices_range&
  tag_invoke(std::tag_invoke::vertices_fn_t, const graph_type& g) {
      return g.vertices_;
  }
  vertices_range vertices_;
}
```

csr_partite_graph? Working Idea

```
template <class EV      = void,
          class VV      = void,
          class GV      = void,
          integral VId = uint32_t,
          class Alloc   = allocator<uint32_t>>
class csr_partite_graph;


using G = csr_partite_graph<std::variant<double,double,void>, // 3 partitions
                            std::variant<int,double,bool>,    // 3 partitions
                            void>;
template<class G>
vertex_range_t<G> partition(g, size_t n);
template <class G>
size_t partition_size(g);
```

rr_adaptor Graph Container

```cpp
template <range_of_ranges Outer, std::ranges::random_access_range VVR>
requires std::ranges::contiguous_range<Outer>
class rr_adaptor {
  template <std::ranges::forward_range ERng, class EProj = std::identity>
  rr_adaptor(VVR& vertex_values,
             const ERng& erng,
             const EProj& eproj = EProj(),
             bool dup_edges = false);
};

using city_names_type = vector<city_name_type>;
struct route {
  city_id_type target_id = 0;
  double       distance  = 0.0; // km
};
using RR                   = std::vector<std::list<route>>;
using routes_rr_graph_type = rr_adaptor<RR, city_names_type>;
```

dynamic_graph Graph Container

```
template <class EV      = void,
          class VV      = void,
          class GV      = void,
          bool  Sourced = false,
          class VId     = uint32_t,
          class Traits  = vofl_graph_traits<EV, VV, GV, Sourced, VId>>
class dynamic_graph;
```

dynamic_graph Traits

```cpp
template <class EV, class VV, class GV, bool Sourced, class VId>
struct vofl_graph_traits {
  using edge_value_type                    = EV;
  using vertex_value_type                  = VV;
  using graph_value_type                   = GV;
  using vertex_id_type                     = VId;
  constexpr inline const static bool sourced = Sourced;

  using edge_type   = dynamic_edge<EV, VV, GV, Sourced, VId, vofl_graph_traits>;
  using vertex_type = dynamic_vertex<EV, VV, GV, Sourced, VId, vofl_graph_traits>;
  using graph_type  = dynamic_graph<EV, VV, GV, Sourced, VId, vofl_graph_traits>;

  using vertices_type = vector<vertex_type>;
  using edges_type    = forward_list<edge_type>;
};

using Traits = vofl_graph_traits<double, std::string, std::string>;
using G      = dynamic_adjacency_graph<Traits>;
```

WRAP-UP

Timeline

Final Comments

Find Out More

Timeline

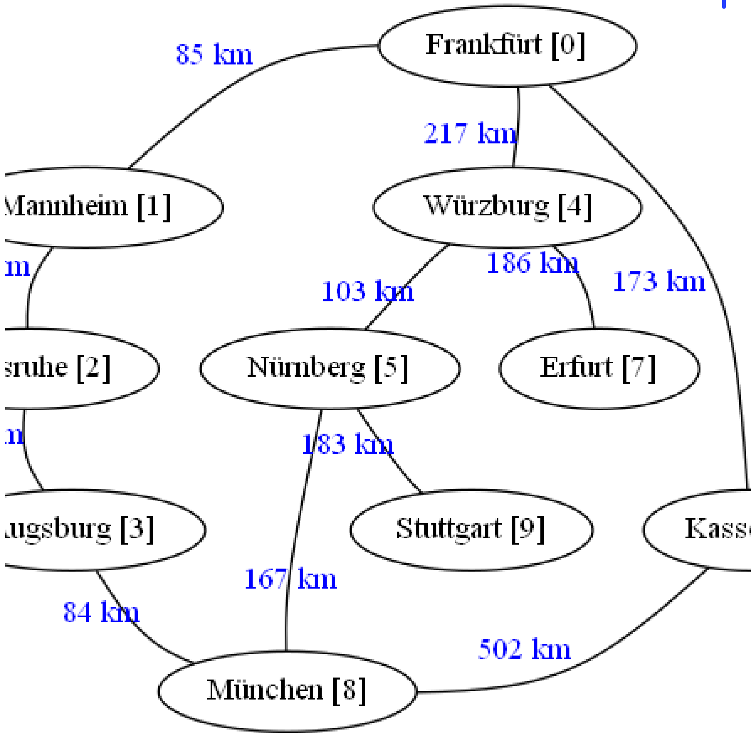
- Acceptance of P1709r3 by SG19 Machine Learning by December
  - Matching implementation in graph-v2
- Review and acceptance by SG6 Numerics
- Review and acceptance by SG14 Game, Embedded, Low Latency
- Review and acceptance by Library Evolution Working Group
- Acceptance into the standard by Feb-2025 for C++2c

Final Comments

- Parallel algorithms aren’t being addressed yet
- Input welcome
- Experience even more welcome
- Recommended to wait for end of year to review
- “star” the GitHub repository to help us

To Find Out More

- Standard Proposal
  - Code: https://github.com/stdgraph/graph-v2
  - Paper: https://wg21.link/P1709r3 (or in graph-v2/doc/latex)
  - This presentation in graph-v2/example/CppCon2022
- NW Graph
  - Code: https://github.com/pnnl/NWGraph
  - Paper: NWGraph: A Library of Generic Graph Algorithms and Data Structures in C++20
- CppCon 2021: Generic Graph Libraries in C++20



Contact
- phil.ratzloff@sas.com
- andrew.lumsdaine@tiledb.com