Pulling Visitors

Inverting Visitor-Based Control Flow

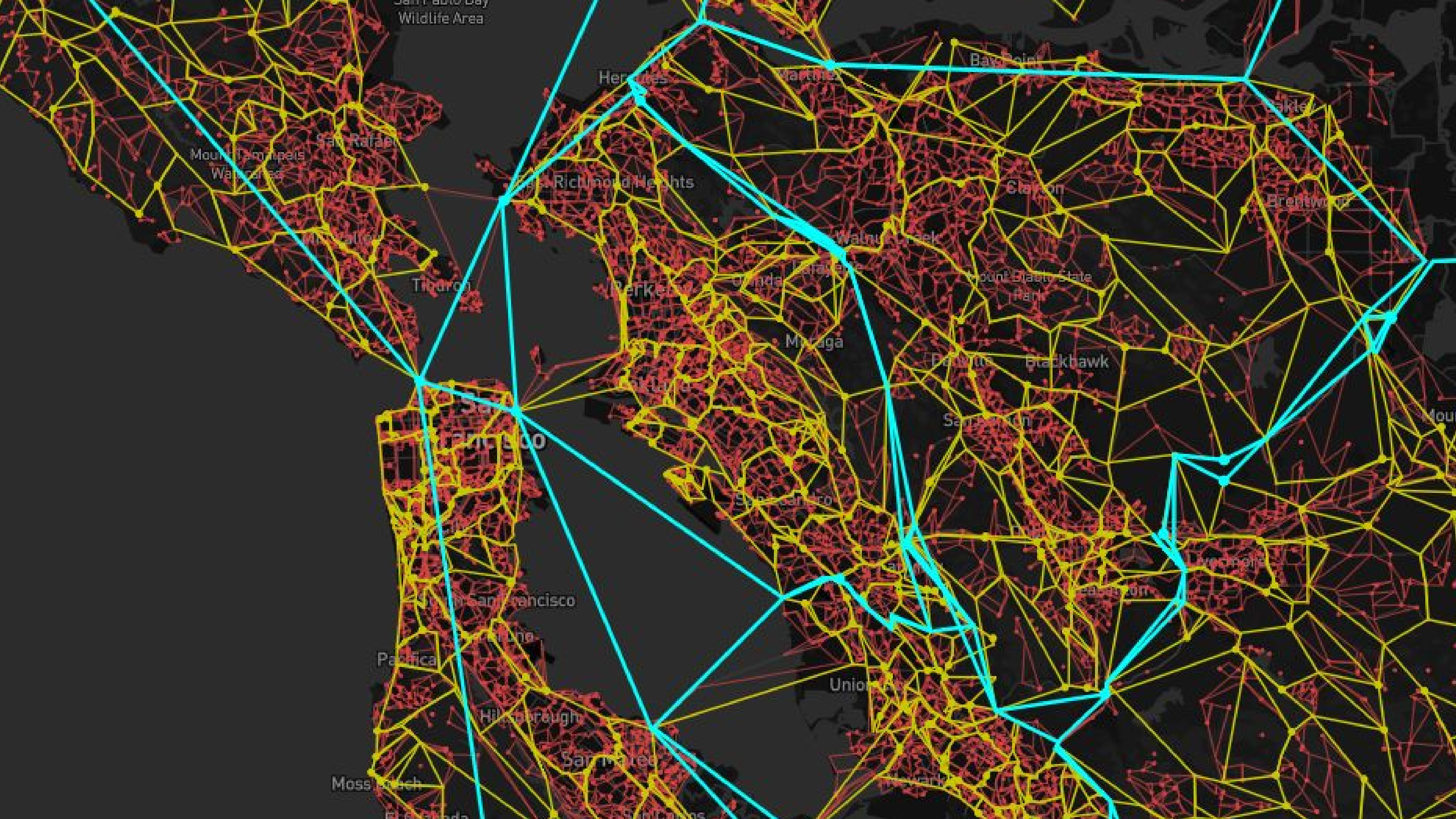
Agenda

Daniel J H, works for Mapbox on Graphs

Boost.Graph introduction, from visitors to iterators

https://github.com/daniel-j-h/cppnow2016 (git.io/cppnow2016-bgl)

Nat Goodspeed, works for Linden Lab on Second Life
Boost.Coroutine problem solution and gory details


San Pablo Bay
Wildlife Area
Hercules
Martinez
Bay Point
Oakley
San Rafael
Mount Tamalpais
Watershed
Mill Valley
Tiburon
East Richmond Heights
Clayton
Brentwood
Walnut Creek
Lafayette
Orinda
Berkeley
Mount Diablo State
Park
Moraga
Danville
Blackhawk
San Ramon
San
Francisco
San Leandro
South San Francisco
San Bruno
Pacifica
Union City
Pleasanton
Livermore
Hillsborough
San Mateo
Moss Beach
Newark

Boost.Graph’s Generic Building-Blocks

Data structures (graph types)

Iterators (edges, vertices)

Properties (internal, external)

Algorithms (breadth-first search, dijkstra)

Visitors (examine_vertex)

Graph Concepts

Graph types are models for Graph Concepts, determines functionality

IncidenceGraph (source, target, out_edges)

BidirectionalGraph (in_edges)

VertexListGraph (vertices)

Graph Representations

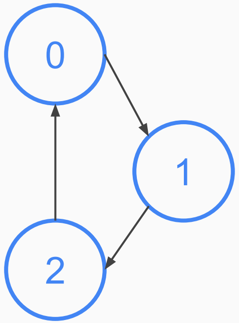
Customizable through template tags: Directed, Undirected, Properties

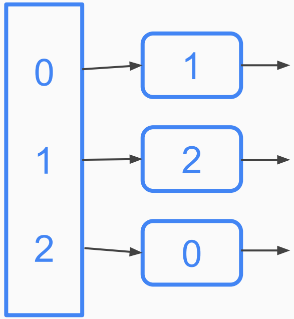
adjacency_list

adjacency_matrix

compressed_sparse_row_graph

adjacency_list<vecS, listS, directedS>

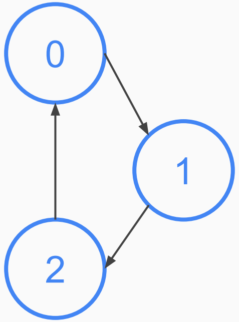


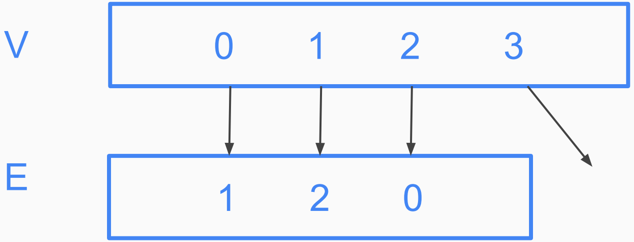


```
using graph_t = adjacency_list<vecS, vecS, directedS>;


graph_t graph(3);


add_edge(0, 1, graph);
add_edge(1, 2, graph);
add_edge(2, 0, graph);
```

```
struct edge_data_t { int duration = 0; };
using graph_t = adjacency_list<vecS, vecS, directedS, no_property, edge_data_t>;
graph_t graph(3);
add_edge(0, 1, edge_data_t{100}, graph);

auto duration = [&graph](auto edge) { return graph[edge].duration; };
auto positive = [](auto duration) { return duration > 0; };

auto d = accumulate(edges(graph) | transformed(duration) | filtered(positive), 0);
```

compressed_sparse_row_graph<directedS>





V 0 1 2 3

E 1 2 0

```
edges(v):
   first = E[V[v]];
   last  = E[V[v + 1]];
```

```
using graph_t = compressed_sparse_row_graph<directedS>;
using vertex_t = graph_traits<graph_t>::vertex_descriptor;

vector<vertex_t> sources{0, 1, 2};
vector<vertex_t> targets{1, 2, 0};

auto tag = construct_inplace_from_sources_and_targets;
graph_t graph{tag, sources, targets, 3};
```

Graph Algorithms

Visitors provide algorithm customization points

Graph Walk (bfs, dfs)

Shortest Paths (dijkstra, a-star)

Max-Flow / Min-Cut (edmonds_karp_max_flow)

```
struct discover_visitor : default_bfs_visitor {
    void discover_vertex(const vertex_t vertex, const graph_t&) {
        cout << vertex << endl;
    }
};


vertex_t source{0};
breadth_first_search(graph, source, visitor(discover_visitor{}));
```

Use-Case Bidirectional Dijkstra

Baseline router to compare against

Start first search on graph from source

Start second search on reversed graph from target

Step both searches (ping-pong) until they meet in the middle

Problem: how to stop and resume visitors

```
vertex_t middle;


async(dijkstra_shortest_path(graph, source, visitor(ping_pong{middle});

async(dijkstra_shortest_path(rev_graph, target, visitor(ping_pong{middle});
```

Coroutines for Cooperative Multitasking

Bind coroutine to visitor, get lazy Dijkstra generator for free

No explicit synchronization, no threads (concurrency != parallelism)

Aha Moment: can be stopped, can be resumed, proper iterators (stdlib)

Technique works for all visitors, and especially well for Boost.Graph

```
using coro_t = coroutines::asymmetric_coroutine<vertex_t>;

struct dijkstra_stepwise : default_dijkstra_visitor {
    dijkstra_stepwise(coro_t::push_type& sink_) : sink(sink_) {}

    void examine_vertex(const vertex_t vertex, const graph_t&) const {
        sink(vertex);
    }
    coro_t::push_type& sink;
};
```

```
coro_t::pull_type lazy_forward_vertices{[&](auto& sink) {
    dijkstra_shortest_paths_no_color_map(graph, source,
        weight_map(get(&edge_data_t::distance, graph))
        .predecessor_map(forward_prev_map)
        .visitor(dijkstra_stepwise{sink}));
}};

while (lazy_forward_vertices && lazy_backward_vertices)
    // lazy_forward_vertices.get();  lazy_forward_vertices();
```

```
auto poi = [&graph](auto vertex) { return has_poi(vertex, graph); };
auto it = find_if(lazy_forward_vertices, poi);


if (it != end(lazy_forward_vertices))
    std::cout << *it << std::endl;
```

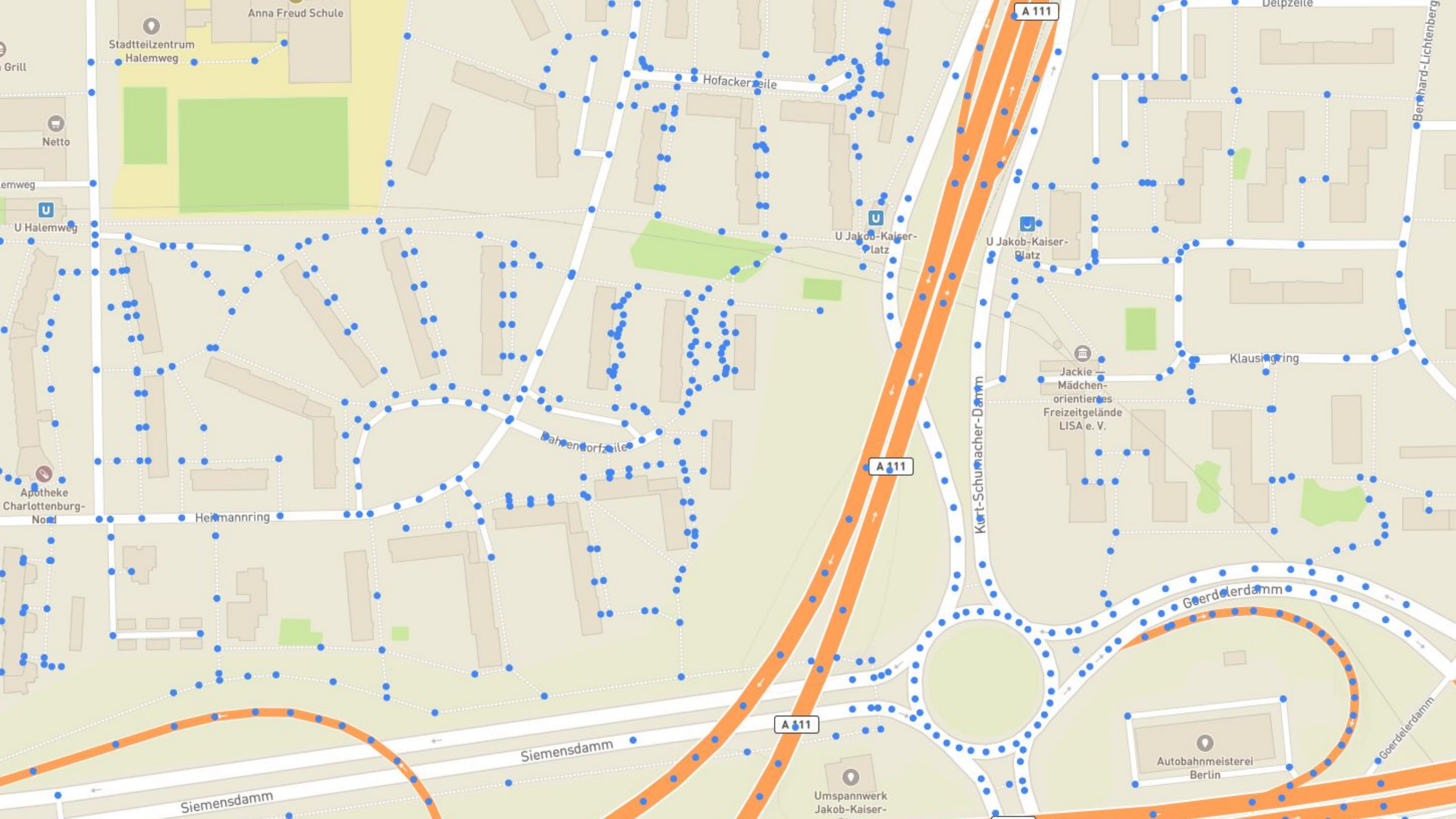
Give Boost.Graph a try!

OpenStreetMap (extract nodes and ways: libosmium)

Construct graph, add properties (location, Boost.Geometry Haversine distance)

Route on the graph (Boost.Geometry’s RTree for initial coordinate lookup)

Visualize the graph (simplification: tippecanoe)


Anna Freud Schule
Stadtteilzentrum Halemweg
Grill
Netto
lemweg
U Halemweg
Apotheke Charlottenburg-Nord
Heilmannring
Hofackerzeile
Bahrendorfzeile
U Jakob-Kaiser-Platz
U Jakob-Kaiser-Platz
A 111
A 111
A 111
Delpzeile
Bernhard-Lichtenberg
Klausingring
Jackie – Mädchen-orientiertes Freizeitgelände LISA e. V.
Kurt-Schumacher-Damm
Goerdelerdamm
Goerdelerdamm
Siemensdamm
Siemensdamm
Autobahnmeisterei Berlin
Umspannwerk Jakob-Kaiser-

Our Take Aways

Powerful trio: Boost.Graph + Boost.Range + Boost.Geometry

Switching to 32 bit vertex and edge index types in CSR (size_t default)

Parallel Boost.Graph vs. r3.4xlarge (120 GB RAM), r3.8xlarge (250 GB RAM)

Boost.Graph + Boost.Coroutine: from visitors to generators, stdlib integration

Generic EventVisitors

Specific visitor

```
using coro_t = coroutines::asymmetric_coroutine<vertex_t>;
struct dijkstra_stepwise : default_dijkstra_visitor {
    dijkstra_stepwise(coro_t::push_type& sink_) : sink(sink_) {}

    void examine_vertex(const vertex_t vertex, const graph_t&) const {
        sink(vertex);
    }
    coro_t::push_type& sink;
};
```

Specific visitor

```
using coro_t = coroutines::asymmetric_coroutine<vertex_t>;
struct dijkstra_stepwise : default_dijkstra_visitor {
    dijkstra_stepwise(coro_t::push_type& sink_) : sink(sink_) {}

    void examine_vertex(const vertex_t vertex, const graph_t&) const {
        sink(vertex);
    }
    coro_t::push_type& sink;
};
```

Specific visitor

```
using coro_t = coroutines::asymmetric_coroutine<vertex_t>;
struct dijkstra_stepwise : default_dijkstra_visitor {
    dijkstra_stepwise(coro_t::push_type& sink_) : sink(sink_) {}

    void examine_vertex(const vertex_t vertex, const graph_t&) const {
        sink(vertex);
    }
    coro_t::push_type& sink;
};
```

Specific visitor

```
using coro_t = coroutines::asymmetric_coroutine<vertex_t>;
struct dijkstra_stepwise : default_dijkstra_visitor {
    dijkstra_stepwise(coro_t::push_type& sink_) : sink(sink_) {}

    // ...
};

dijkstra_shortest_paths(graph, source,
                        visitor(dikjstra_stepwise{sink}));
```

Specific visitor

```
using coro_t = coroutines::asymmetric_coroutine<vertex_t>;
struct dijkstra_stepwise : default_dijkstra_visitor {
    dijkstra_stepwise(coro_t::push_type& sink_) : sink(sink_) {}

    // ...
};

dijkstra_shortest_paths(graph, source,
                        visitor(dikjstra_stepwise{sink}));
```

EventVisitor mechanism

```
struct dijkstra_stepwise {
    // part of the EventVisitor API
    typedef boost::on_examine_vertex event_filter;

    explicit dijkstra_stepwise(coro_t::push_type& sink_) : sink{sink_} {}

    void operator()(const vertex_t vertex, const graph_t&) { sink(vertex); }

    coro_t::push_type& sink;
};
```

EventVisitor mechanism

```
struct dijkstra_stepwise {
  // part of the EventVisitor API
  typedef boost::on_examine_vertex event_filter;

  // ...

};

dijkstra_shortest_paths(graph, source,
                        visitor(boost::make_dijkstra_visitor(
                                dijkstra_stepwise{sink}));
```

EventVisitor mechanism

```
template <typename Tag>
struct dijkstra_stepwise {
    // part of the EventVisitor API
    typedef Tag event_filter;

    explicit dijkstra_stepwise(coro_t::push_type& sink_) : sink{sink_} {}

    void operator()(const vertex_t vertex, const Graph&) { sink(vertex); }

    coro_t::push_type& sink;
};
```

EventVisitor mechanism

```
template <typename Tag>
struct dijkstra_stepwise {
    typedef Tag event_filter;

    // ...

};

dijkstra_shortest_paths(graph, source,
                                        visitor(boost::make_dijkstra_visitor(
                                                    dijkstra_stepwise<boost::on_examine_vertex>{sink})));
```

EventVisitor mechanism

```
template <typename Tag>
struct dijkstra_stepwise {
    typedef Tag event_filter;

    // ...

};

template <typename Tag>
make_dijkstra_stepwise(coro_t::push_type& sink_, Tag) {
    return dijkstra_stepwise<Tag>(sink_);
}
```

EventVisitor mechanism

```
template <typename Tag>
make_dijkstra_stepwise(coro_t::push_type& sink_, Tag) {
  return dijkstra_stepwise<Tag>(sink_);
}

dijkstra_shortest_paths(graph, source,
                               visitor(boost::make_dijkstra_visitor(
                                      make_dijkstra_stepwise(sink, boost::on_examine_vertex()))));
```

CoroEventVisitor mechanism

```
template <typename EdgeOrVertex, typename Tag>
class CoroEventVisitorBase
{
public:
        // required by EventVisitor API
        typedef Tag event_filter;
        typedef typename boost::coroutines::asymmetric_coroutine<EdgeOrVertex>::push_type coro_t;
        CoroEventVisitorBase(coro_t& sink):
           mSink(sink)
        {}
protected:
        coro_t& mSink;
};
```

CoroEventVisitor mechanism

```
template <typename EdgeOrVertex, typename Tag>
class CoroEventVisitorBase
{
public:
        // required by EventVisitor API
        typedef Tag event_filter;
        typedef typename boost::coroutines::asymmetric_coroutine<EdgeOrVertex>::push_type coro_t;
        CoroEventVisitorBase(coro_t& sink):
          mSink(sink)
        {}
protected:
        coro_t& mSink;
};
```

CoroEventVisitor mechanism

```
template <typename EdgeOrVertex, typename Tag>
class CoroEventVisitorBase
{
public:
        // required by EventVisitor API
        typedef Tag event_filter;
        typedef typename boost::coroutines::asymmetric_coroutine<EdgeOrVertex>::push_type coro_t;
        CoroEventVisitorBase(coro_t& sink):
          mSink(sink)
        {}
protected:
        coro_t& mSink;
};
```

CoroEventVisitor mechanism

```
template <typename EdgeOrVertex, typename Tag>
struct CoroEventVisitor:
        public  CoroEventVisitorBase<EdgeOrVertex, Tag>
{
        typedef CoroEventVisitorBase<EdgeOrVertex, Tag> super;
        template <typename Coro>
        CoroEventVisitor(Coro& sink): super(sink) {}
        template <typename Graph>
        void operator()(EdgeOrVertex eu, const Graph&) {
           super::mSink(eu);
        }
};
```

CoroEventVisitor mechanism

```
template <typename EdgeOrVertex, typename Tag>
struct CoroEventVisitor:
          public  CoroEventVisitorBase<EdgeOrVertex, Tag>
{
          typedef CoroEventVisitorBase<EdgeOrVertex, Tag> super;
          template <typename Coro>
          CoroEventVisitor(Coro& sink): super(sink) {}
          template <typename Graph>
          void operator()(EdgeOrVertex eu, const Graph&) {
              super::mSink(eu);
          }
};
```

CoroEventVisitor mechanism

```
template <typename EdgeOrVertex, typename Tag>
auto make_coro_visitor(boost::coroutines::push_coroutine<EdgeOrVertex>& sink, Tag) {
        return CoroEventVisitor<EdgeOrVertex, Tag>(sink);
}
```

```
dijkstra_shortest_paths(graph, source,
                        visitor(boost::make_dijkstra_visitor(
                                make_coro_visitor(sink, boost::on_examine_vertex()))));
```

CoroEventVisitor mechanism

```
template <typename EdgeOrVertex, typename Tag>
auto make_coro_visitor(boost::coroutines::push_coroutine<EdgeOrVertex>& sink, Tag) {
        return CoroEventVisitor<EdgeOrVertex, Tag>(sink);
}
```

```
dijkstra_shortest_paths(graph, source,
                        visitor(boost::make_dijkstra_visitor(
                            make_coro_visitor(sink, boost::on_examine_vertex()))));
```

CoroEventVisitor mechanism

```
template <typename EdgeOrVertex, typename Tag>
auto make_coro_visitor(typename boost::coroutines::asymmetric_coroutine<EdgeOrVertex>::push_type& sink, Tag) {
        return CoroEventVisitor<EdgeOrVertex, Tag>(sink);
}

dijkstra_shortest_paths(graph, source,
                visitor(boost::make_dijkstra_visitor(
                        make_coro_visitor<vertex_t>(sink, boost::on_examine_vertex()))));
```

CoroEventVisitor mechanism

```
template <typename EdgeOrVertex, typename Tag>
auto make_coro_visitor(typename boost::coroutines::asymmetric_coroutine<EdgeOrVertex>::push_type& sink, Tag) {
	return CoroEventVisitor<EdgeOrVertex, Tag>(sink);
}
```

```
dijkstra_shortest_paths(graph, source,
				visitor(boost::make_dijkstra_visitor(
					make_coro_visitor<vertex_t>(sink, boost::on_examine_vertex()))));
```

CoroEventVisitor mechanism

```
namespace boost {
namespace coroutines {

template< typename T >
struct asymmetric_coroutine
{
        typedef push_coroutine< T > push_type;
        typedef pull_coroutine< T > pull_type;
};

}}
```

CoroEventVisitor mechanism

```
using graph_t = boost::compressed_sparse_row_graph<boost::directedS>;
using vertex_t = typename boost::graph_traits<graph_t>::vertex_descriptor;
using edge_t = typename boost::graph_traits<graph_t>::edge_descriptor;
using coro_t = boost::coroutines::asymmetric_coroutine<vertex_t>;

coro_t::pull_type generator{[&](coro_t::push_type& sink) { //
  dijkstra_shortest_paths(graph, source, visitor(boost::make_dijkstra_visitor(
                                          make_coro_visitor(sink,
                                                            boost::on_examine_vertex()))));
}};
```

CoroEventVisitor mechanism

```
// coroutine type producing vertex_t, graph_t tuples

typedef std::tuple<vertex_t, const graph_t&> VertexGraph;

typedef boost::coroutines::asymmetric_coroutine<VertexGraph> vgcoro_t;
```

CoroEventVisitor mechanism

```
template <typename EdgeOrVertex, typename TupleGraph, typename Tag>
struct CoroEventVisitor<std::tuple<EdgeOrVertex, TupleGraph>, Tag>:
        public  CoroEventVisitorBase<std::tuple<EdgeOrVertex, TupleGraph>, Tag>
{
        typedef std::tuple<EdgeOrVertex, TupleGraph> Tuple;
        typedef CoroEventVisitorBase<Tuple, Tag> super;
        template <typename Coro>
        CoroEventVisitor(Coro& sink): super(sink) {}
        template <typename Graph>
        void operator()(EdgeOrVertex eu, const Graph& g) {
            super::mSink(Tuple(eu, g));
        }
};
```

CoroEventVisitor mechanism

```
template <typename EdgeOrVertex, typename TupleGraph, typename Tag>
struct CoroEventVisitor<std::tuple<EdgeOrVertex, TupleGraph>, Tag>:
        public  CoroEventVisitorBase<std::tuple<EdgeOrVertex, TupleGraph>, Tag>
{
        typedef std::tuple<EdgeOrVertex, TupleGraph> Tuple;
        typedef CoroEventVisitorBase<Tuple, Tag> super;
        template <typename Coro>
        CoroEventVisitor(Coro& sink): super(sink) {}
        template <typename Graph>
        void operator()(EdgeOrVertex eu, const Graph& g) {
           super::mSink(Tuple(eu, g));
        }
};
```

CoroEventVisitor mechanism

```
template <typename EdgeOrVertex, typename TupleGraph, typename Tag>
struct CoroEventVisitor<std::tuple<EdgeOrVertex, TupleGraph>, Tag>:
        public  CoroEventVisitorBase<std::tuple<EdgeOrVertex, TupleGraph>, Tag>
{
        typedef std::tuple<EdgeOrVertex, TupleGraph> Tuple;
        typedef CoroEventVisitorBase<Tuple, Tag> super;
        template <typename Coro>
        CoroEventVisitor(Coro& sink): super(sink) {}
        template <typename Graph>
        void operator()(EdgeOrVertex eu, const Graph& g) {
            super::mSink(Tuple(eu, g));
        }
};
```

CoroEventVisitor mechanism

```
// coroutine type producing vertex_t, graph_t tuples
typedef std::tuple<vertex_t, const graph_t&> VertexGraph;
typedef boost::coroutines::asymmetric_coroutine<VertexGraph> vgcoro_t;

vgcoro_t::pull_type generator{[&](vgcoro_t::push_type& sink) { //
    dijkstra_shortest_paths(graph, source, visitor(boost::make_dijkstra_visitor(
                                               make_coro_visitor(sink,
                                                                   boost::on_examine_vertex())))));
}};
```

CoroEventVisitor mechanism

```
template <typename EdgeOrVertex, typename TupleGraph, typename Tag>
struct CoroEventVisitor<std::tuple<EdgeOrVertex, TupleGraph>, Tag>:
        public  CoroEventVisitorBase<std::tuple<EdgeOrVertex, TupleGraph>, Tag>
{
        typedef std::tuple<EdgeOrVertex, TupleGraph> Tuple;
        typedef CoroEventVisitorBase<Tuple, Tag> super;
        template <typename Coro>
        CoroEventVisitor(Coro& sink): super(sink) {}
        template <typename Graph>
        void operator()(EdgeOrVertex eu, const Graph& g) {
           super::mSink(Tuple(eu, g));
        }
};
```

CoroEventVisitor mechanism

```
template <typename EdgeOrVertex, typename TupleGraph, typename Tag>
struct CoroEventVisitor<std::tuple<EdgeOrVertex, TupleGraph>, Tag>:
        public  CoroEventVisitorBase<std::tuple<EdgeOrVertex, TupleGraph>, Tag>
{
        typedef std::tuple<EdgeOrVertex, TupleGraph> Tuple;
        typedef CoroEventVisitorBase<Tuple, Tag> super;
        template <typename Coro>
        CoroEventVisitor(Coro& sink): super(sink) {}
        template <typename Graph>
        void operator()(EdgeOrVertex eu, const Graph& g) {
           super::mSink(Tuple(eu, g));
        }
};
```

EventVisitor redux

```
boost::dijkstra_shortest_paths(G, s,
   boost::make_dijkstra_visitor(
    std::make_pair(boost::record_distances(d, boost::on_tree_edge()),
    std::make_pair(boost::record_predecessors(p.begin(),
                                               boost::on_tree_edge()),
                   copy_graph(G_copy, boost::on_examine_edge())))) );
```

EventVisitor redux

```
boost::dijkstra_shortest_paths(G, s,
  boost::make_dijkstra_visitor(
   std::make_pair(boost::record_distances(d, boost::on_tree_edge()),
   std::make_pair(boost::record_predecessors(p.begin(),
                            boost::on_tree_edge()),
            copy_graph(G_copy, boost::on_examine_edge()))))) );
```

EventVisitor redux

```
boost::dijkstra_shortest_paths(G, s,
  boost::make_dijkstra_visitor(
   std::make_pair(boost::record_distances(d, boost::on_tree_edge()),
   std::make_pair(boost::record_predecessors(p.begin(),
                                                         boost::on_tree_edge()),
                  copy_graph(G_copy, boost::on_examine_edge())))) );
```

EventVisitor redux

```
boost::dijkstra_shortest_paths(G, s,
  boost::make_dijkstra_visitor(
   std::make_pair(boost::record_distances(d, boost::on_tree_edge()),
   std::make_pair(boost::record_predecessors(p.begin(),
                          boost::on_tree_edge()),
           copy_graph(G_copy, boost::on_examine_edge())))) );
```

EventVisitor redux

```
boost::dijkstra_shortest_paths(G, s,
  boost::make_dijkstra_visitor(
   std::make_pair(boost::record_distances(d, boost::on_tree_edge()),
   std::make_pair(boost::record_predecessors(p.begin(),
                              boost::on_tree_edge()),
            copy_graph(G_copy, boost::on_examine_edge())))) );
```

EventVisitor redux

```
boost::dijkstra_shortest_paths(G, s,
  boost::make_dijkstra_visitor(
   std::make_pair(boost::record_distances(d, boost::on_tree_edge()),
   std::make_pair(boost::record_predecessors(p.begin(),
                          boost::on_tree_edge()),
           copy_graph(G_copy, boost::on_examine_edge()))) );
```

EventVisitor detour

```
boost::dijkstra_shortest_paths(G, s,
   boost::make_dijkstra_visitor(
    std::make_pair(boost::record_distances(d, boost::on_tree_edge()),
    std::make_pair(boost::record_predecessors(p.begin(),
                                              boost::on_tree_edge()),
                   copy_graph(G_copy, boost::on_examine_edge())))) );
```

EventVisitor detour

```
template <typename EventVisitor>
auto evisitors(EventVisitor visitor) {
        return visitor;
}

template <typename EventVisitor, typename... EventVisitors>
auto evisitors(EventVisitor visitor, EventVisitors... rest) {
        return std::make_pair(visitor, evisitors(rest...));
}

template <typename... EventVisitors>
auto make_dijkstra_visitor(EventVisitors... visitors) {
        return boost::make_dijkstra_visitor(evisitors(visitors...));
}
```

EventVisitor detour

```
boost::dijkstra_shortest_paths(G, s,
  make_dijkstra_visitor(
    boost::record_distances(d, boost::on_tree_edge()),
    boost::record_predecessors(p.begin(), boost::on_tree_edge()),
    copy_graph(G_copy, boost::on_examine_edge())) );
```

CoroEventVisitor mechanism

```
boost::dijkstra_shortest_paths(G, s,
  make_dijkstra_visitor(
    make_coro_visitor(sink, boost::on_discover_vertex()),
    make_coro_visitor(sink, boost::on_examine_vertex()),
    make_coro_visitor(sink, boost::on_finish_vertex())) );
```

CoroEventVisitor mechanism

std::type_index(typeid(boost::on_examine_vertex))

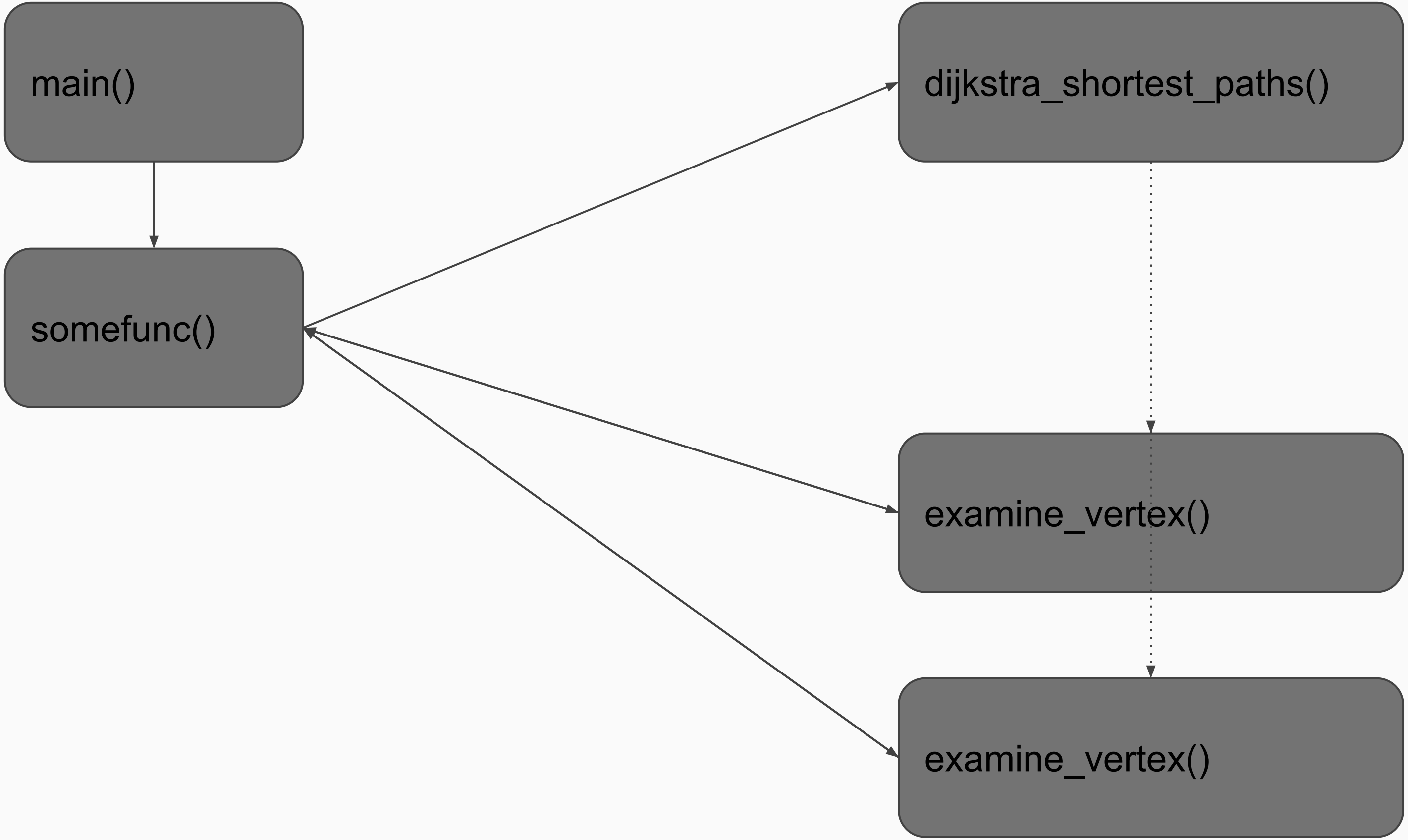
CoroEventVisitor mechanism

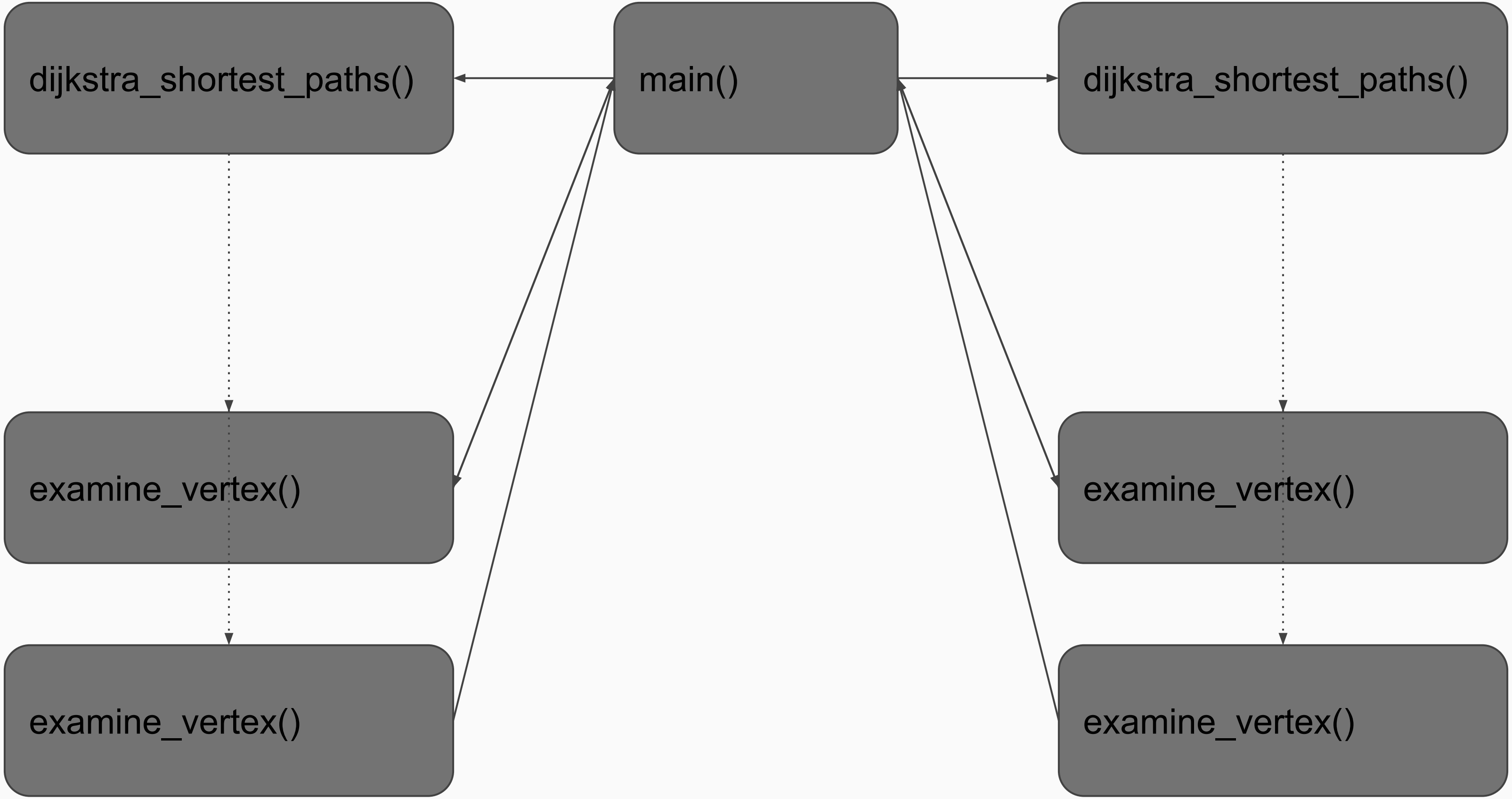
```
template <typename EdgeOrVertex, typename TupleGraph, typename Tag>
struct CoroEventVisitor<std::tuple<EdgeOrVertex, TupleGraph, std::type_index>, Tag>:
  public  CoroEventVisitorBase<std::tuple<EdgeOrVertex, TupleGraph, std::type_index>, Tag>
{
        typedef std::tuple<EdgeOrVertex, TupleGraph, std::type_index> Tuple;
        typedef CoroEventVisitorBase<Tuple, Tag> super;
        template <typename Coro>
        CoroEventVisitor(Coro& sink): super(sink) {}
        template <typename Graph>
        void operator()(EdgeOrVertex eu, const Graph& g) {
           super::mSink(Tuple(eu, g, std::type_index(typeid(Tag))));
        }
};
```

But how do you get away with that?
Coroutines and Stacks

Each coroutine runs on its own stack

- Stack depth in opaque algorithm doesn’t matter


main()
somefunc()
dijkstra_shortest_paths()
examine_vertex()
examine_vertex()



dijkstra_shortest_paths()

main()

dijkstra_shortest_paths()

examine_vertex()

examine_vertex()

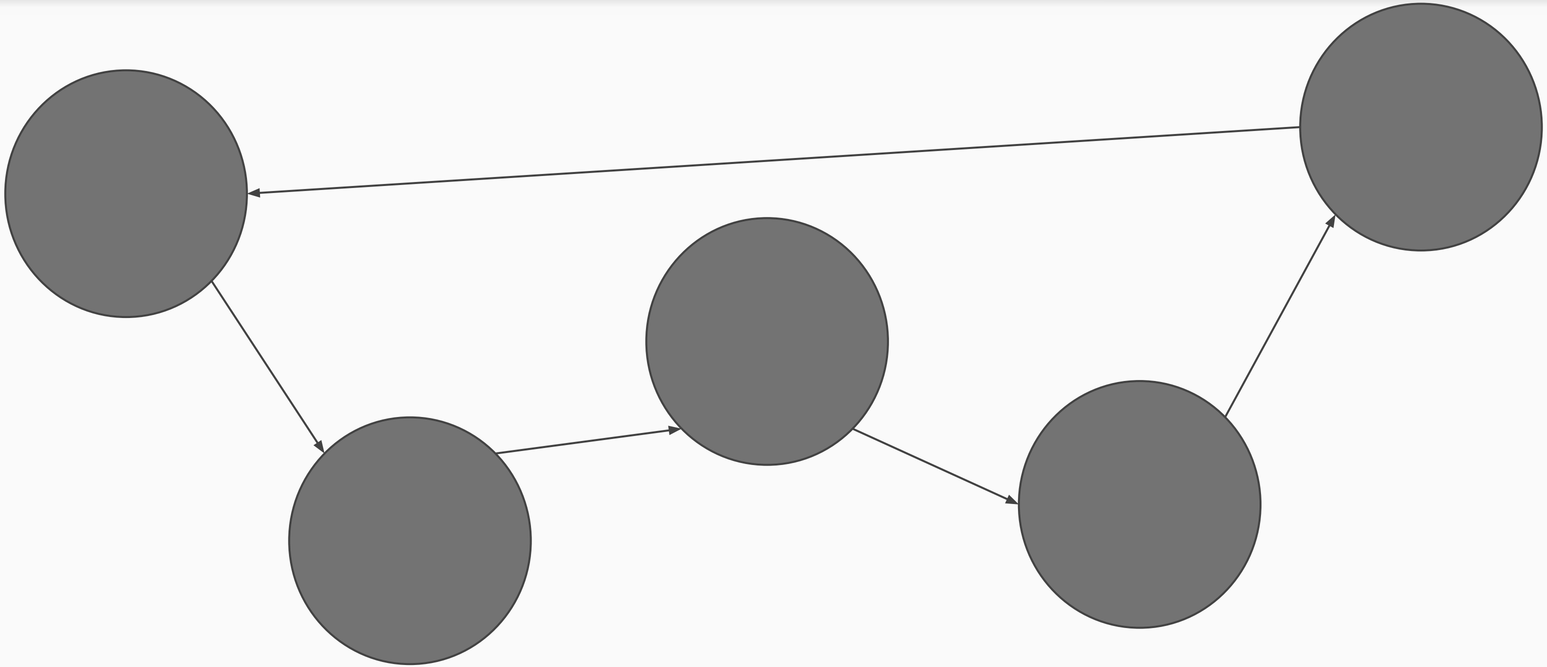
examine_vertex()

examine_vertex()

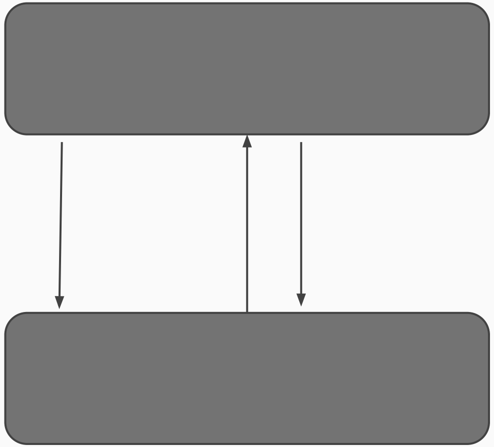
Boost.Coroutine

- Symmetric coroutines
- Asymmetric coroutines

Symmetric Coroutines



Asymmetric Coroutines



push_type, pull_type

```
boost::coroutines::asymmetric_coroutine<int>::push_type mycoro(
        [](boost::coroutines::asymmetric_coroutine<int>::pull_type& source) {
                ...
        });
```

push_type, pull_type

```
boost::coroutines::asymmetric_coroutine<int>::push_type mycoro(
        [](boost::coroutines::asymmetric_coroutine<int>::pull_type& source) {
                ...
        });
```

push_type, pull_type

```
boost::coroutines::asymmetric_coroutine<int>::push_type mycoro(
    [](boost::coroutines::asymmetric_coroutine<int>::pull_type& source) {
        ...
    });
```

push_type, pull_type

```
boost::coroutines::asymmetric_coroutine<int>::pull_type mycoro(
    [](boost::coroutines::asymmetric_coroutine<int>::push_type& sink) {
        …
    });
```

push_type, pull_type

```
boost::coroutines::asymmetric_coroutine<int>::pull_type mycoro(
        [](boost::coroutines::asymmetric_coroutine<int>::push_type& sink) {
                sink(17);
        });
```

push_type, pull_type

```
boost::coroutines::asymmetric_coroutine<int>::pull_type mycoro(...);

while (mycoro) {
    int foo = mycoro.get();
    mycoro();
}
```

push_type, pull_type

```
boost::coroutines::asymmetric_coroutine<int>::pull_type mycoro(...);

while (mycoro) {
    int foo = mycoro.get();
    mycoro();
}
```

push_type, pull_type

```
boost::coroutines::asymmetric_coroutine<int>::pull_type mycoro(...);

while (mycoro) {
    int foo = mycoro.get();
    mycoro();
}
```

push_type, pull_type

```
boost::coroutines::asymmetric_coroutine<int>::pull_type mycoro(...);

for (int foo : mycoro) {
    // ...
}
```

This tactic is applicable to *any* library whose API involves callbacks or visitors.

Questions?

https://gist.github.com/nat-goodspeed/