Boost.Graph tutorial

Richel Bilderbeek

December 10, 2015

Contents

1 Introduction

I needed this tutorial in 2006, when I started experimenting with Boost.Graph. More specifically, I needed a tutorial that:

- Orders concepts chronologically
- Increases complexity gradually
- Shows complete pieces of code

What I had were the book [2] and the Boost.Graph website, both did not satisfy these requirements.

1.1 Code snippets

For every concept, I will show

- the 'do' function: a function that achieves a goal, for example 'create_empty_undirected_graph'
- the 'demo' function: a function that demonstrates how to call the first, for example 'demonstrate_create_empty_undirected_graph'

I enjoy to show concepts by putting those in (long-named) functions. These functions sometimes border the trivial, by, for example, only calling a Boost.Graph function. On the other hand, these functions have more English-sounding names, resulting in demonstration code that is readable.

All coding snippets are taken from compiled C++ code.

1.2 Coding style

I use the coding style from the Core C++ Guidelines. At the time of this writing, the Core C++ Guidelines were still in early development, so I can only hope the conventions I then chose to follow are still Good Ideas.

I prefer to use the keyword auto over doubling the lines of code for using statements. Because the 'do' functions return an explicit data type, these can be used for reference. If you really want to know a type, you can use the 'get_type_name' function (chapter 11.1). On the other hand, I am explicit of which data types I choose: I will prefix the types by thir namespace, so to distinguish between types like 'std::array' and 'boost::array'.

1.3 Feedback

I have tried hard to strictly follow the style as described above. If you find I deviated from these decisions somewhere, I would be grateful if you'd let know.

2 Building a graph without properties

Boost.Graph is about creating graphs. In this chapter we create the simplest of graphs, in which edges and nodes have no properties (e.g. having a name). We will build:

- An empty (directed) graph, which is the default type: see chapter 2.1
- An empty (undirected) graph: see chapter 2.2
- K_2, an undirected graph with two vertices and one edge, chapter 2.12

In the process, some basic (sometimes bordering trivial) functions are shown:

- Adding a vertex: see chapter 2.3
- Getting all vertices: see chapter 2.5
- Getting all vertex descriptors: see chapter 2.6
- Adding an edge: see chapter 2.7
- Getting all edges: see chapter 2.9
- Getting all edge descriptors: see chapter 2.11

These functions are mostly there for completion and showing which data types are used.

2.1 Creating an empty (directed) graph

Let's create a trivial empty graph!

Algorithm 1 shows the function to create an empty (directed) graph.

Algorithm 1 Creating an empty (directed) graph

```
#include <boost/graph/adjacency_list.hpp>

boost::adjacency_list<>
create_empty_directed_graph() noexcept
{
  return boost::adjacency_list<>();
}
```

Algorithm 2 demonstrates the 'create_empty_directed_graph' function.

Algorithm 2 Demonstration of 'create_empty_directed_graph'

```
#include "create_empty_directed_graph.h"

void create_empty_directed_graph_demo() noexcept
{
  const auto g = create_empty_directed_graph();
}
```

Congratulations, you've just created a boost::adjacency_list with its default template arguments. For your reference, these default template argument denote that you've just created a graph, in which:

- The out edges are stored in a std::vector
- The vertices are stored in a std::vector
- The edges have a direction
- The vertices, edges and graph have no properties
- The edges are stored in a std::list

The boost::adjacency_list is the most commonly used graph type, the other is the boost::adjacency_matrix.

2.2 Creating an empty undirected graph

Let's create another trivial empty graph! This time, we make the graph undirected.

Algorith 3 shows how to create an undirected graph.

Algorithm 3 Creating an empty undirected graph

```
#include <boost/graph/adjacency_list.hpp>

boost::adjacency_list<
  boost::vecS,
  boost::vecS,
  boost::undirectedS
>
create_empty_undirected_graph() noexcept
{
  return boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS
  >();
}
```

Algorithm 4 demonstrates the 'create_empty_undirected_graph' function.

Algorithm 4 Demonstration of 'create_empty_undirected_graph'

```
#include "create_empty_undirected_graph.h"

void create_empty_undirected_graph_demo() noexcept
{
  const auto g = create_empty_undirected_graph();
}
```

Congratulations, you've just created an undirected graph in which:

- The out edges are stored in a std::vector. This way to store out edges is selected by the first 'boost::vecS'
- The vertices are stored in a std::vector. This way to store vertices is selected by the second 'boost::vecS'
- The graph is undirected. This directionality is selected for by the third template argument, 'boost::undirectedS'
- Vertices, edges and graph have no properties
- Edges are stored in a std::list

The difference between directed and undirected graphs is in the edges: in an undirected graph, an edge connects two vertices without any directionality. In a directed graph, an edge goes from a certain vertex, its source, to another (which may actually be the same), its target.

2.3 Add a vertex

Empty graphs are nice, now its time to add a vertex!

To add a vertex to a graph, the boost::add_vertex function is used as shows in algorithm 5.

Algorithm 5 Adding a vertex to a graph

```
#include <boost/graph/adjacency_list.hpp>

template <typename graph>
void add_vertex(graph& g)
{
  boost::add_vertex(g);
}
```

Algorithm 6 shows how to add a vertex to a directed and undirected graph.

Algorithm 6 Adding a vertex to a graph

```
#include "add_vertex.h"
#include "create_empty_directed_graph.h"
#include "create_empty_undirected_graph.h"

void add_vertex_demo() noexcept
{
  auto g = create_empty_undirected_graph();
  add_vertex(g);

  auto h = create_empty_directed_graph();
  add_vertex(h);
}
```

Note that boost::add_vertex (in the 'add_vertex' function) returns a vertex descriptor, which is ignored for now. Vertex descriptors are looked at in more details at the chapter 2.4.

2.4 Vertex descriptors

A vertex descriptor is a handle to a vertex within a graph.

Vertex descriptors can be obtained by:

- dereference a vertex iterator, see chapter 2.6

Vertex descriptors are used to:

- add and edge between two vertices, see chapter 2.7
- obtain properties of vertex a vertex, for example the vertex its out degrees (chapter 23), the vertex its name (chapter 29), or a custom vertex property (chapter 45)

In this tutorial, vertex descriptors have named prefixed with 'vd_', for example 'vd_1'.

2.5 Get the vertices

You cannot get the vertices. This may sound unexpected, as it must be possible to work on the vertices of a graph. Working on the vertices of a graph is done throught these steps:

- Obtain a vertex iterator pair from the graph
- Dereference a vertex iterator to obtain a vertex descriptor

boost::vertices is used to obtain a vertex iterator pair, as shown in algorithm 7. The first vertex iterator points to the first vertex (its descriptor, to be precise), the second points to beyond the last vertex. In this tutorial, vertex iterator pairs have named prefixed with 'vip_', for example 'vip_1'.

Algorithm 7 Get the vertex iterators of a graph

```
#include <boost/graph/adjacency_list.hpp>

template <class graph>
std::pair<
  typename graph::vertex_iterator,
  typename graph::vertex_iterator
>
get_vertices(const graph& g)
{
  return boost::vertices(g);
}
```

These vertex iterators can be dereferenced to obtain the vertex descriptors. Note that 'get_vertices' will not be used often in isolation: usually one obtains the vertex descriptors immediatly. Just for your references, algorithm 8 demonstrates of the 'get_vertices' function, by showing that the vertex iterators of an empty graph point to the same location.

Algorithm 8 Demonstration of 'get_vertices'

```
#include <cassert>
#include "create_empty_directed_graph.h"
#include "create_empty_undirected_graph.h"
#include "get_vertices.h"

void get_vertices_demo() noexcept
{
  const auto g = create_empty_undirected_graph();
  const auto vip_g = get_vertices(g);
  assert(vip_g.first == vip_g.second);

  const auto h = create_empty_directed_graph();
  const auto vip_h = get_vertices(h);
  assert(vip_h.first == vip_h.second);
}
```

2.6 Get all vertex descriptors

Vertex descriptors are the way to manipulate those vertices. Let's get the all!

Vertex descriptors are obtained from dereferencing vertex iterators. Algorithm 9 shows how to obtain all vertex descriptors from a graph.

Algorithm 9 Get all vertex descriptors of a graph

```
#include <vector>
#include "boost/graph/graph_traits.hpp"

template <class graph>
std::vector<
  typename boost::graph_traits<graph>::vertex_descriptor
> get_vertex_descriptors(const graph& g) noexcept
{
  using boost::graph_traits;
  std::vector<
    typename graph_traits<graph>::vertex_descriptor
  > v;
  for (auto vi = vertices(g);
    vi.first != vi.second;
    ++vi.first)
  {
    v.emplace_back(*vi.first);
  }
  return v;
}
```

The 'get_vertex_descripors' function shows an important concept of the Boost.Graph library: boost::vertices returns two vertex iterators, which in turn can be dereferenced to obtain the vertex descriptors. Algorithm 10 demonstrates that an empty graph has no vertex descriptors.

Algorithm 10 Demonstration of 'get_vertex_descriptors'

```
#include "create_empty_directed_graph.h"
#include "create_empty_undirected_graph.h"
#include "get_vertex_descriptors.h"

void get_vertex_descriptors_demo() noexcept
{
  const auto g = create_empty_undirected_graph();
  const auto vds_g = get_vertex_descriptors(g);
  assert(vds_g.empty());

  const auto h = create_empty_directed_graph();
  const auto vds_h = get_vertex_descriptors(h);
  assert(vds_h.empty());
}
```

2.7 Add an edge

To add an edge to a graph, two vertex descriptors are needed. A vertex descriptor is a handle to the vertex with in graph (vertex descriptors are looked at in more details in chapter 2.4). Algorithm 11 adds two vertices to a graph, and connects these two using boost::add_edge:

Algorithm 11 Adding (two vertices and) an edge to a graph

```
#include <boost/graph/adjacency_list.hpp>

#include <cassert>

template <typename graph>
void add_edge(graph& g)
{
  const auto vd_a = boost::add_vertex(g);
  const auto vd_b = boost::add_vertex(g);
  const auto aer = boost::add_edge(
    vd_a,
    vd_b,
    g
  );

  assert(aer.second);
}
```

This algorithm only shows how to add an isolated edge to a graph, instead of allowing for graphs with higher connectivities. The function boost::add_vertex returns a vertex descriptor, which I prefix with 'vd'. The function boost::add_edge returns a std::pair, consisting of an edge descriptor and a boolean success indicator. In algorithm 11 we assert that this insertion was successfull. Insertion can fail if an edge is already present and duplicates are not allowed.

A demonstration of add_edge is shown in algorith 12, in which an edge is added to both a directed and undirected graph.

Algorithm 12 Demonstration of add_edge

```
#include "add_edge.h"
#include "create_empty_directed_graph.h"
#include "create_empty_undirected_graph.h"

void add_edge_demo() noexcept
{
  auto g = create_empty_undirected_graph();
  add_edge(g);

  auto h = create_empty_directed_graph();
  add_edge(h);
}
```

2.8 boost::add_edge result

When using the function 'boost::add_edge', a 'std::pair<edge_descriptor,bool>' is returned. It contains both the edge descriptor (see chapter 2.10) and a boolean indicating insertion success.

In this tutorial, boost::add_edge results have named prefixed with 'aer_', for example 'aer_1'.

2.9 Getting the edges

You cannot get the edges directly. Working on the edges of a graph is done throught these steps:

- Obtain an edge iterator pair from the graph
- Dereference an edge iterator to obtain an edge descriptor

boost::edges is used to obtain an edge iterator pair. The first edge iterator points to the first edge (its descriptor, to be precise), the second points to beyond the last edge. In this tutorial, edge iterator pairs have named prefixed with 'eip_', for example 'eip_1'.

Algorithm 13 Get the edge iterators of a graph

```
#include <boost/graph/adjacency_list.hpp>

template <class graph>
std::pair<
  typename graph::edge_iterator,
  typename graph::edge_iterator
>
get_edges(const graph& g)
{
  return boost::edges(g);
}
```

These edge iterators can be dereferenced to obtain the edge descriptors. Note that this function will not be used often in isolation: usually one obtains the edge descriptors immediatly.

Algorithm 14 demonstrates 'get_edges' by showing that both iterators of the edge iterator pair point to the same location, when the graph is empty.

Algorithm 14 Demonstration of get_edges

```
#include <cassert>
#include "create_empty_directed_graph.h"
#include "create_empty_undirected_graph.h"
#include "get_edges.h"

void get_edges_demo() noexcept
{
  const auto g = create_empty_undirected_graph();
  const auto eip_g = get_edges(g);
  assert(eip_g.first == eip_g.second);

  auto h = create_empty_directed_graph();
  const auto eip_h = get_edges(h);
  assert(eip_h.first == eip_h.second);
}
```

2.10 Edge descriptors

An edge descriptor is a handle to an edge within a graph. Edge descriptors are used to:

- obtain the name, or other properties, of an edge

In this tutorial, edge descriptors have named prefixed with 'ed_', for example 'ed_1'.

2.11 Get all edge descriptors

Obtaining all edge descriptors is not as simple of a function as you'd guess:

Algorithm 15 Get all edge descriptors of a graph

```
#include <vector>
#include "boost/graph/graph_traits.hpp"

template <class graph>
std::vector<
  typename boost::graph_traits<graph>::edge_descriptor
> get_edge_descriptors(const graph& g) noexcept
{
  using boost::graph_traits;
  std::vector<
    typename graph_traits<graph>::edge_descriptor
  > v;
  for (auto vi = edges(g);
    vi.first != vi.second;
    ++vi.first)
  {
    v.emplace_back(*vi.first);
  }
  return v;
}
```

This does show an important concept of the Boost.Graph library: boost::edges returns to vertex iterators, that can be dereferenced to obtain the vertex descriptors.

Algorithm 16 demonstrates the 'get_edge_descriptor', by showing that empty graphs do not have any edge descriptors.

Algorithm 16 Demonstration of get_edge_descriptors

```
#include <cassert>
#include "create_empty_directed_graph.h"
#include "create_empty_undirected_graph.h"
#include "get_edge_descriptors.h"

void get_edge_descriptors_demo() noexcept
{
  const auto g = create_empty_directed_graph();
  const auto eds_g = get_edge_descriptors(g);
  assert(eds_g.empty());

  const auto h = create_empty_undirected_graph();
  const auto eds_h = get_edge_descriptors(h);
  assert(eds_h.empty());
}
```

2.12 Creating K_2, a fully connected graph with two vertices

Finally, we are going to create a graph!

To create a fully connected graph with two vertices (also called K_2), one needs two vertices and one (undirected) edge, as depicted in figure 1.

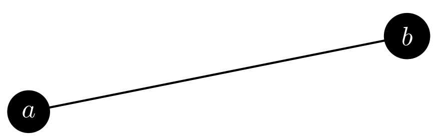

Figure 1: K_2: a fully connected graph with two vertices named a and b

To create K_2, the following code can be used:

Algorithm 17 Creating K_2 as depicted in figure 1

```
#include "create_empty_undirected_graph.h"

boost::adjacency_list<
  boost::vecS,
  boost::vecS,
  boost::undirectedS
>
create_k2_graph() noexcept
{
  auto g = create_empty_undirected_graph();
  const auto vd_a = boost::add_vertex(g);
  const auto vd_b = boost::add_vertex(g);
  const auto aer = boost::add_edge(vd_a, vd_b, g);
  assert(aer.second);
  return g;
}
```

To save defining the type, we call the 'create_empty_undirected_graph' function. The vertex descriptors (see chapter 2.4) created by two boost::add_vertex calls are stored to add an edge to the graph. From boost::add_edge its return type (see chapter 2.8), it is only checked that insertion has been successfull.

Note that the graph lacks all properties: nodes do not have names, nor do edges.

Algorithm 18 demonstrates how to 'create_k2_graph' and uses all functions currently described by this tutorial.

Algorithm 18 Demonstration of 'create_k2_graph'

```
#include <cassert>
#include <iostream>

#include "create_k2_graph.h"
#include "get_edge_descriptors.h"
#include "get_edges.h"
#include "get_vertex_descriptors.h"
#include "get_vertices.h"

void create_k2_graph_demo() noexcept
{
  const auto g = create_k2_graph();
  const auto vip = get_vertices(g);
  assert(vip.first != vip.second);
  const auto vds = get_vertex_descriptors(g);
  assert(vds.size() == 2);
  const auto eip = get_edges(g);
  assert(eip.first != eip.second);
  const auto eds = get_edge_descriptors(g);
  assert(eds.size() == 1);
}
```

3 Measuring simple traits of a graph without properties

Graphs without edge and vertex properties have plenty of things to measure. Additionally, it allows you to test abd debug your code.

3.1 Counting the number of vertices

Use boost::num_vertices, as shown here:

Algorithm 19 Count the numbe of vertices

```
#include <boost/graph/adjacency_list.hpp>

///Get the number of vertices a graph has
template <class graph>
int get_n_vertices(const graph& g)
{
  return static_cast<int>(boost::num_vertices(g));
}
```

The function 'get_n_vertices' is demonstrated in algorithm 20, to measure the number of vertices of an empty (zero) and K_2 (two) graph.

Algorithm 20 Demonstration of the 'get_n_vertices' function

```
#include "get_n_vertices.h"

#include <cassert>

#include "create_empty_directed_graph.h"
#include "create_k2_graph.h"
#include "get_n_vertices.h"

void get_n_vertices_demo() noexcept
{
  const auto g = create_empty_directed_graph();
  assert(get_n_vertices(g) == 0);

  const auto h = create_k2_graph();
  assert(get_n_vertices(h) == 2);
}
```

3.2 Counting the number of edges

Use boost::num_edges, as shown here:

Algorithm 21 Count the number of edges

```
#include <boost/graph/adjacency_list.hpp>

///Get the number of edges a graph has
template <class graph>
int get_n_edges(const graph& g)
{
  return static_cast<int>(boost::num_edges(g));
}
```

The function 'get_n_edges' is demonstrated in algorithm 22, to measure the number of vertices of an empty (zero) and K_2 (one) graph.

Algorithm 22 Demonstration of the 'get_n_edges' function

```
#include "get_n_edges.h"

#include <cassert>

#include "create_empty_directed_graph.h"
#include "create_k2_graph.h"

void get_n_edges_demo() noexcept
{
  const auto g = create_empty_directed_graph();
  assert(get_n_edges(g) == 0);

  const auto h = create_k2_graph();
  assert(get_n_edges(h) == 1);
}
```

3.3 Getting the vertices' out degree

The out degree of a vertex is the number of edges that originate at it.

Algorithm 23 Get the vertices' out degrees

```
#include <vector>

template <typename graph>
std::vector<int> get_vertex_out_degrees(const graph& g)
{
  std::vector<int> v;
  for (auto p = vertices(g);
    p.first != p.second;
    ++p.first) {
    v.emplace_back(out_degree(*p.first,g));
  }
  return v;
}
```

The out degrees of the vertices are obtained directly from the vertex descriptor and then put into a std::vector. Note that the std::vector has element type 'int', instead of 'graph::degree_size_type', as one should prefer using int (over unsigned int) in an interface [1][1]. Also, avoid using an unsigned an int for the sake of gaining that one more bit [3][2].

Albeit K_2 is a simple graph, we can use it to demonstrate 'get_vertex_out_degrees' on, as shown in algorithm 24.

Algorithm 24 Demonstration of the 'get_vertex_out_degrees' function

```
#include <cassert>

#include "create_k2_graph.h"
#include "get_vertex_out_degrees.h"

void get_vertex_out_degrees_demo() noexcept
{
  const auto g = create_k2_graph();
  const std::vector<int> expected_out_degrees{1,1};
  const std::vector<int> vertex_out_degrees{
      get_vertex_out_degrees(g)};
  assert(expected_out_degrees == vertex_out_degrees);
}
```

[1] Chapter 9.2.2
[2] Chapter 4.4

4 Building graphs with built-in properties

Up until now, the graphs created have had edges and vertices without any propery. In this chapter, graphs will be created, in which edges vertices can have a (std::string) name and/or are of a custom type.

- An empty (undirected) graph that allows for vertices with names: see chapter 4.1
- $K_2$with named vertices: see chapter 4.4
- An empty (undirected) graph that allows for edges and vertices with names: see chapter 4.5
- $K_3$with named edges and vertices: see chapter 4.8
- An empty (undirected) graph that allows for custom vertices: see chapter 6.1
- $K_2$with custom vertices: see chapter 6.4
- An empty (undirected) graph that allows for custom edges and vertices: see chapter 6.5
- $K_3$with custom edges and vertices: see chapter 6.7

In the process, some basic (sometimes bordering trivial) functions are shown:

- Adding a named vertex: see chapter 4.2
- Getting the vertices' names: see chapter 4.3
- Adding an named edge: see chapter 4.6
- Adding a custom vertex: see chapter 6.2
- Adding a custom edge: see chapter 6.6

These functions are mostly there for completion and showing which data types are used.

4.1 Creating an empty graph with named vertices

Let's create a trivial empty graph, in which the vertices can have a name:

Algorithm 25 Creating an empty graph with named vertices

```
#include <boost/graph/adjacency_list.hpp>

boost::adjacency_list<
  boost::vecS,
  boost::vecS,
  boost::undirectedS,
  boost::property<
    boost::vertex_name_t,std::string
  >
>
create_empty_named_vertices_graph() noexcept
{
  return boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    boost::property<
      boost::vertex_name_t,std::string
    >
  >();
}
```

This graph:

- has its out edges stored in a std::vector (due to the first boost::vecS)
- has its vertices stored in a std::vector (due to the second boost::vecS)
- is undirected (due to the boost::undirectedS)
- The vertices have one property: they have a name, that is of data type std::string (due to the boost::property< boost::vertex_name_t,std::string>')
- Edges and graph have no properties
- Edges are stored in a std::list

The boost::adjacency_list has a new, fourth template argument 'boost::property< boost::vertex_name_t,std::string>'. This can be read as: "vertices have the property 'boost::vertex_name_t', that is of data type 'std::string"'. Or simply: "vertices have a name that is stored as a std::string".

Algorithm 26 shows how to create this graph. Note that all the earlier functions defined in this tutorial keep working as expected.

Algorithm 26 Demonstration if the 'create_empty_named_vertices_graph' function

```
#include <cassert>

#include "create_empty_named_vertices_graph.h"
#include "get_edge_descriptors.h"
#include "get_edges.h"
#include "get_vertex_descriptors.h"
#include "get_vertices.h"

void create_empty_named_vertices_graph_demo() noexcept
{
  const auto g = create_empty_named_vertices_graph();
  const auto vip = get_vertices(g);
  assert(vip.first == vip.second);
  const auto vds = get_vertex_descriptors(g);
  assert(vds.empty());
  const auto eip = get_edges(g);
  assert(eip.first == eip.second);
  const auto eds = get_edge_descriptors(g);
  assert(eds.empty());
}
```

4.2 Add a vertex with a name

Adding a vertex without a name was trivially easy (see chapter 5). Adding a vertex with a name is less easy:

Algorithm 27 Add a vertex with a name

```
#include <boost/graph/adjacency_list.hpp>

template <typename graph>
void add_named_vertex(graph& g, const std::string& name)
{
  const auto vd_a = boost::add_vertex(g);
  auto vertex_name_map = boost::get(boost::vertex_name,g)
      ;
  vertex_name_map[vd_a] = name;
}
```

Instead of calling 'boost::add_vertex' with an additional argument containing the name of the vertex[3] , multiple things need to be done. When adding a new vertex to the graph, the vertex descriptor (as describes in chapter 2.4) is stored. After obtaining the name map from the graph (using 'boost::get(boost::vertex_name,g)'), the name of the vertex is set using that vertex descriptor.

Using add_named_vertex is straightforward, as demonstrated by algorithm 28.

Algorithm 28 Demonstration of 'add_named_vertex'

```
#include <cassert>

#include "add_named_vertex.h"
#include "create_empty_named_vertices_graph.h"
#include "get_vertex_descriptors.h"

void add_named_vertex_demo() noexcept
{
  auto g = create_empty_named_vertices_graph();
  add_named_vertex(g,"Lex");
  assert(get_vertex_descriptors(g).size() == 1);
}
```

4.3 Getting the vertices' names

When the vertices of a graph have named vertices, one can extract them as such:

[3] I am unsure if this would have been a good interface. I am sure I expected this interface myself. I do see a problem with multiple properties and the order of initialization, but initialization can follow the same order as the the property list.

Algorithm 29 Get the vertices' names

```
#include <string>
#include <vector>
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/properties.hpp>

//TODO: generalize to return any type
template <typename graph>
std::vector<std::string> get_vertex_names(const graph& g)
{
  std::vector<std::string> v;

  const auto vertex_name_map = get(boost::vertex_name,g);

  for (auto p = vertices(g);
    p.first != p.second;
    ++p.first) {
    v.emplace_back(get(vertex_name_map, *p.first));
  }
  return v;
}
```

The names of the vertices are obtained from a boost::property_map and then put into a std::vector. Note that the std::vector has element type 'std::string', instead of extracting the type from the graph. If you know how to do so, please email me.

When trying to get the vertices' names from a graph without vertices with names, you will get the error 'formed reference to void' (see chapter 12.1).

Algorithm 30 shows how to add two named vertices and how to get their names.

Algorithm 30 Demonstration of 'get_vertex_names'

```
#include <cassert>

#include "add_named_vertex.h"
#include "create_empty_named_vertices_graph.h"
#include "get_vertex_names.h"

void get_vertex_names_demo() noexcept
{
  auto g = create_empty_named_vertices_graph();
  const std::string vertex_name_1{"Chip"};
  const std::string vertex_name_2{"Chap"};
  add_named_vertex(g,vertex_name_1);
  add_named_vertex(g,vertex_name_2);
  const std::vector<std::string> expected_names{
      vertex_name_1, vertex_name_2};
  const std::vector<std::string> vertex_names{
      get_vertex_names(g)};
  assert(expected_names == vertex_names);
}
```

4.4 Creating K_2 with named vertices

We extend K_2 of chapter 2.12 by naming the vertices 'from' and 'to', as depicted in figure 2:

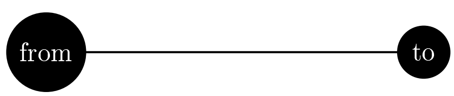

Figure 2: K_2: a fully connected graph with two vertices with the text 'from' and 'to'

To create K_2, the following code can be used:

Algorithm 31 Creating K_2 as depicted in figure 2

```
#include "create_named_vertices_k2_graph.h"
#include "create_empty_named_vertices_graph.h"

boost::adjacency_list<
  boost::vecS,
  boost::vecS,
  boost::undirectedS,
  boost::property<boost::vertex_name_t,std::string>
>
create_named_vertices_k2_graph() noexcept
{
  auto g = create_empty_named_vertices_graph();
  const auto vd_a = boost::add_vertex(g);
  const auto vd_b = boost::add_vertex(g);
  const auto aer = boost::add_edge(
    vd_a,
    vd_b,
    g
  );
  assert(aer.second);

  auto name_map = boost::get(boost::vertex_name,g);
  name_map[vd_a] = "from";
  name_map[vd_b] = "to";

  return g;
}
```

Most of the code is a repeat of algorithm 17. In the end, the names are obtained as a boost::property_map and set.

Also the demonstration code (algorithm) is very similar to the demonstration code of the create_k2_graph function ().

Algorithm 32 Demonstrating the 'create_k2_graph' function

```
#include <cassert>

#include "create_named_vertices_k2_graph.h"
#include "get_vertex_names.h"

void create_named_vertices_k2_graph_demo() noexcept
{
  const auto g = create_named_vertices_k2_graph();
  const std::vector<std::string> expected_names{"from", "
      to"};
  const std::vector<std::string> vertex_names =
      get_vertex_names(g);
  assert(expected_names == vertex_names);
}
```

4.5 Creating an empty graph with named edges and vertices

Let's create a trivial empty graph, in which the both the edges and vertices can have a name:

Algorithm 33 Creating an empty graph with named edges and vertices

```
#include <string>
#include <boost/graph/adjacency_list.hpp>

boost::adjacency_list<
  boost::vecS,
  boost::vecS,
  boost::undirectedS,
  boost::property<boost::vertex_name_t,std::string>,
  boost::property<boost::edge_name_t,std::string>
>
create_empty_named_edges_and_vertices_graph() noexcept
{
  return boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    boost::property<
      boost::vertex_name_t,std::string
    >,
    boost::property<
      boost::edge_name_t,std::string
    >
  >();
}
```

This graph:

- has its out edges stored in a std::vector (due to the first boost::vecS)
- has its vertices stored in a std::vector (due to the second boost::vecS)
- is undirected (due to the boost::undirectedS)
- The vertices have one property: they have a name, that is of data type std::string (due to the boost::property< boost::vertex_name_t,std::string>')
- The edges have one property: they have a name, that is of data type std::string (due to the boost::property< boost::edge_name_t,std::string>')
- The graph has no properties
- Edges are stored in a std::list

The boost::adjacency_list has a new, fifth template argument 'boost::property< boost::edge_name_t,std::string>'. This can be read as: "edges have the property 'boost::edge_name_t', that is of data type 'std::string"'. Or simply: "edges have a name that is stored as a std::string".

Algorithm 34 shows how to create this graph. Note that all the earlier functions defined in this tutorial keep working as expected.

Algorithm 34 Demonstration if the 'create_empty_named_edges_and_vertices_graph' function

```
#include <cassert>

#include "add_named_edge.h"
#include "create_empty_named_edges_and_vertices_graph.h"
#include "get_edge_names.h"
#include "get_vertex_names.h"

void create_empty_named_edges_and_vertices_graph_demo()
    noexcept
{
  auto g = create_empty_named_edges_and_vertices_graph();
  add_named_edge(g,"Reed");
  const std::vector<std::string> expected_vertex_names{""
      ,""};
  const std::vector<std::string> vertex_names =
      get_vertex_names(g);
  assert(expected_vertex_names == vertex_names);
  const std::vector<std::string> expected_edge_names{"
      Reed"};
  const std::vector<std::string> edge_names =
      get_edge_names(g);
  assert(expected_edge_names == edge_names);
}
```

4.6 Adding a named edge

Adding an edge with a name:

Algorithm 35 Add a vertex with a name

```
#include <boost/graph/adjacency_list.hpp>

#include <cassert>

template <typename graph>
void add_named_edge(graph& g, const std::string&
    edge_name)
{
  const auto vd_a = boost::add_vertex(g);
  const auto vd_b = boost::add_vertex(g);
  const auto aer = boost::add_edge(vd_a, vd_b, g);
  assert(aer.second);

  auto edge_name_map = boost::get(boost::edge_name,g);
  edge_name_map[aer.first] = edge_name;
}
```

In this code snippet, the edge descriptor (see chapter 2.10 if you need to refresh your memory) when using 'boost::add_edge' is used as a key to change the edge its name map.

The algorithm 36 shows how to add a named edge to an empty graph. When trying to add named vertices to graph without this property, you will get the error 'formed reference to void' (see chapter 12.1).

Algorithm 36 Demonstration of the 'add_named_edge' function

```
#include <cassert>

#include "add_named_edge.h"
#include "create_empty_named_edges_and_vertices_graph.h"
#include "get_n_edges.h"

void add_named_edge_demo() noexcept
{
  auto g = create_empty_named_edges_and_vertices_graph();
  add_named_edge(g,"Richards");
  assert(get_n_edges(g) == 1);
}
```

4.7 Getting the edges' names

When the edges of a graph have named vertices, one can extract them as such:

Algorithm 37 Get the edges' names

```
#include <string>
#include <vector>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/properties.hpp>

//TODO: generalize to return any type
template <typename graph>
std::vector<std::string> get_edge_names(const graph& g)
{
  std::vector<std::string> v;

  const auto edge_name_map = get(boost::edge_name,g);

  for (auto p = boost::edges(g);
    p.first != p.second;
    ++p.first) {
    v.emplace_back(get(edge_name_map, *p.first));
  }
  return v;
}
```

The names of the edges are obtained from a boost::property_map and then put into a std::vector. The algorithm 38 shows how to apply this function.

Would you dare to try to get the edges' names from a graph without vertices with names, you will get the error 'formed reference to void' (see chapter 12.1).

Algorithm 38 Demonstration of the 'get_edge_names' function

```
#include <cassert>

#include "add_named_edge.h"
#include "create_empty_named_edges_and_vertices_graph.h"
#include "get_edge_names.h"

void get_edge_names_demo() noexcept
{
  auto g = create_empty_named_edges_and_vertices_graph();
  const std::string edge_name_1{"Eugene"};
  const std::string edge_name_2{"Another Eugene"};
  add_named_edge(g, edge_name_1);
  add_named_edge(g, edge_name_2);
  const std::vector<std::string> expected_names{
      edge_name_1, edge_name_2};
  const std::vector<std::string> edge_names{
      get_edge_names(g)};
  assert(expected_names == edge_names);
}
```

4.8 Creating K_3 with named edges and vertices

We extend the graph K_2 with named vertices of chapter 4.4 by adding names to the edges, as depicted in figure 3:

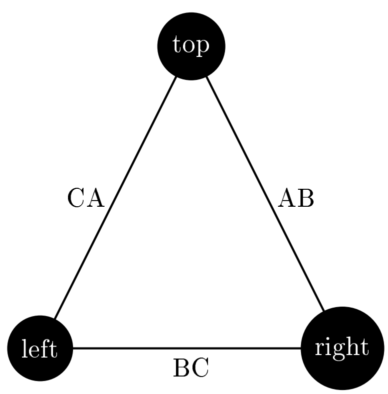

Figure 3: K_3: a fully connected graph with three named edges and vertices

To create K_3, the following code can be used:

Algorithm 39 Creating K_3 as depicted in figure 3

```
#include <boost/graph/adjacency_list.hpp>
#include <string>

boost::adjacency_list<
  boost::vecS,
  boost::vecS,
  boost::undirectedS,
  boost::property<boost::vertex_name_t,std::string>,
  boost::property<boost::edge_name_t,std::string>
>
create_named_edges_and_vertices_k3_graph() noexcept
{
  auto g = create_empty_named_edges_and_vertices_graph();
  const auto vd_a = boost::add_vertex(g);
  const auto vd_b = boost::add_vertex(g);
  const auto vd_c = boost::add_vertex(g);
  const auto aer_ab = boost::add_edge(vd_a, vd_b, g);
  assert(aer_ab.second);
  const auto aer_bc = boost::add_edge(vd_b, vd_c, g);
  assert(aer_bc.second);
  const auto aer_ca = boost::add_edge(vd_c, vd_a, g);
  assert(aer_ca.second);

  //Add vertex names
  auto vertex_name_map = boost::get(boost::vertex_name,g)
    ;
  vertex_name_map[vd_a] = "top";
  vertex_name_map[vd_b] = "right";
  vertex_name_map[vd_c] = "left";

  //Add edge names
  auto edge_name_map = boost::get(boost::edge_name,g);
  edge_name_map[aer_ab.first] = "AB";
  edge_name_map[aer_bc.first] = "BC";
  edge_name_map[aer_ca.first] = "CA";

  return g;
}
```

Most of the code is a repeat of algorithm 31. In the end, the edge names are obtained as a boost::property_map and set. Algorithm 40 shows how to create the graph and measure its edge and vertex names.

Algorithm 40 Demonstration of the 'create_named_edges_and_vertices_k3' function

```
#include <cassert>
#include <iostream>
#include "create_named_edges_and_vertices_k3_graph.h"
#include "get_edge_names.h"
#include "get_vertex_names.h"

void create_named_edges_and_vertices_k3_graph_demo()
    noexcept
{
  const auto g = create_named_edges_and_vertices_k3_graph
      ();
  const std::vector<std::string> expected_vertex_names{"
      top", "right", "left"};
  const std::vector<std::string> vertex_names{
      get_vertex_names(g)};
  assert(expected_vertex_names == vertex_names);
  const std::vector<std::string> expected_edge_names{"AB"
      , "BC", "CA"};
  const std::vector<std::string> edge_names{
      get_edge_names(g)};
  assert(expected_edge_names == edge_names);
}
```

5 Measuring simple graphs traits of a graph with named edges and vertices

Measuring simple traits of the graphs created allows

5.1 Count vertex name

count_vertices_with_name

5.2 Find a vertex by its name

find_vertex_with_name

5.3 Get a named vertex its in-degree

get_named_vertex_in_degree

- degree_size_type in_degree(vertex_descriptor u, const adjacency_list& g) . Returns the in-degree of a vertex

5.4 Get a named vertex its out-degree

get_named_vertex_out_degree

- degree_size_type in_degree(vertex_descriptor u, const adjacency_list& g) . Returns the in-degree of a vertex

6 Building graphs with custom properties

Up until now, the graphs created have had edges and vertices with the built-in name propery. In this chapter, graphs will be created, in which the edges and vertices can have a custom 'my_edge' and 'my_edge' type[4].

- An empty (undirected) graph that allows for custom vertices: see chapter 6.1
- $K_2$with custom vertices: see chapter 6.4
- An empty (undirected) graph that allows for custom edges and vertices: see chapter 6.5
- $K_3$with custom edges and vertices: see chapter 6.7

In the process, some basic (sometimes bordering trivial) functions are shown:

- Adding a custom vertex: see chapter 6.2
- Adding a custom edge: see chapter 6.6

These functions are mostly there for completion and showing which data types are used.

6.1 Create an empty graph with custom vertices

Say we want to use our own vertex class as graph nodes. This is done in multiple steps:

1. Create a custom vertex class, called 'my_vertex'
2. Install a new property, called 'vertex_custom_type'
3. Use the new property in creating a boost::adjacency_list

[4]I do not intend to be original in naming my data types

6.1.1 Creating the custom vertex class

In this example, I create a custom vertex class. Here I will show the header file of it, as the implementation of it is not important yet.

Algorithm 41 Declaration of my_vertex

```
#ifndef MY_VERTEX_H
#define MY_VERTEX_H

#include <string>

class my_vertex
{
public:
  my_vertex(
    const std::string& name = "",
    const std::string& description = "",
    const double x = 0.0,
    const double y = 0.0
  ) noexcept;
  std::string m_name;
  std::string m_description;
  double m_x;
  double m_y;
};

bool operator==(const my_vertex& lhs, const my_vertex&
   rhs) noexcept;

#endif // MY_VERTEX_H
```

my_vertex is a class that has multiple properties: two doubles 'm_x' ('m_' stands for member) and 'm_y', and two std::strings m_name and m_description. my_vertex is copyable, but cannot trivially be converted to a std::string.

6.1.2 Installing the new property

Installing a new property would have been easier, if 'more C++ compilers were standards conformant' ([2], chapter 3.6, footnote at page 52). Boost.Graph uses the BOOST_INSTALL_PROPERTY macro to allow using a custom property:

Algorithm 42 Installing the vertex_custom_type property

```
#include <boost/graph/properties.hpp>

namespace boost {
  enum vertex_custom_type_t { vertex_custom_type = 314 };
  BOOST_INSTALL_PROPERTY(vertex,custom_type);
}
```

The enum value 314 must be unique.

6.1.3 Create the empty graph with custom vertices

Algorithm 43 Creating an empty graph with custom vertices

```
#include <boost/graph/adjacency_list.hpp>
#include "install_vertex_custom_type.h"
#include "my_vertex.h"

boost::adjacency_list<
  boost::vecS,
  boost::vecS,
  boost::undirectedS,
  boost::property<
    boost::vertex_custom_type_t,my_vertex
  >
>
create_empty_custom_vertices_graph() noexcept
{
  return boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    boost::property<
      boost::vertex_custom_type_t,my_vertex
    >
  >();
}
```

This graph:

- has its out edges stored in a std::vector (due to the first boost::vecS)
- has its vertices stored in a std::vector (due to the second boost::vecS)

- is undirected (due to the boost::undirectedS)
- The vertices have one property: they have a custom type, that is of data type my_vertex (due to the boost::property< boost::vertex_custom_type_t,my_vertex>')
- The edges and graph have no properties
- Edges are stored in a std::list

The boost::adjacency_list has a new, fourth template argument 'boost::property< boost::vertex_custom_type_t,my_vertex>'. This can be read as: "vertices have the property 'boost::vertex_custom_type_t', which is of data type 'my_vertex"'. Or simply: "vertices have a custom type called my_vertex".

6.2 Add a custom vertex

Adding a custom vertex is very similar to adding a named vertex (chapter 4.2).

Algorithm 44 Add a custom vertex

```
#include <boost/graph/adjacency_list.hpp>
#include "install_vertex_custom_type.h"
#include "my_vertex.h"

template <typename graph>
void add_custom_vertex(graph& g, const my_vertex& v)
{
  const auto vd_a = boost::add_vertex(g);
  const auto my_vertex_map = boost::get(boost::
      vertex_custom_type,g);
  my_vertex_map[vd_a] = v;
}
```

When having added a new (abstract) vertex to the graph, the vertex descriptor is used to set the my_vertex in the graph its my_vertex map (using 'boost::get(boost::vertex_custom_type,g)').

6.3 Getting the vertices' my_vertexes[5]

When the vertices of a graph have any associated my_vertex, one can extract these as such:

[5]the name 'my_vertexes' is chosen to indicate this function returns a container of my_vertex

Algorithm 45 Get the vertices' my_vertexes

```
#include <vector>
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/properties.hpp>
#include "install_vertex_custom_type.h"
#include "my_vertex.h"

//TODO: generalize to return any type
template <typename graph>
std::vector<my_vertex> get_vertex_my_vertexes(const graph
    & g)
{
  std::vector<my_vertex> v;

  const auto my_vertexes_map = get(boost::
      vertex_custom_type,g);

  for (auto p = vertices(g);
    p.first != p.second;
    ++p.first) {
    v.emplace_back(get(my_vertexes_map, *p.first));
  }
  return v;
}
```

The my_vertex object associated with the vertices are obtained from a boost::property_map and then put into a std::vector.

When trying to get the vertices' my_vertex from a graph without my_vertex objects associated, you will get the error 'formed reference to void' (see chapter 12.1).

6.4 Creating K_2 with custom vertices

We reproduce the K_2 with named vertices of chapter 4.4 , but with our custom vertices intead:

Algorithm 46 Creating K_2 as depicted in figure 2

```
#include "install_vertex_custom_type.h"
#include "my_vertex.h"
#include "create_empty_custom_vertices_graph.h"

#include <boost/graph/adjacency_list.hpp>

boost::adjacency_list<
  boost::vecS,
  boost::vecS,
  boost::undirectedS,
  boost::property<
    boost::vertex_custom_type_t,my_vertex
  >
>
create_custom_vertices_k2_graph() noexcept
{
  auto g = create_empty_custom_vertices_graph();
  const auto vd_a = boost::add_vertex(g);
  const auto vd_b = boost::add_vertex(g);
  const auto aer = boost::add_edge(vd_a, vd_b, g);
  assert(aer.second);

  //Add names
  auto my_vertex_name_map = boost::get(boost::
      vertex_custom_type,g);
  my_vertex_name_map[vd_a]
    = my_vertex("from","source",0.0,0.0);
  my_vertex_name_map[vd_b]
    = my_vertex("to","target",3.14,3.14);

  return g;
}
```

Most of the code is a slight modification of algorithm 31. In the end, the my_vertices are obtained as a boost::property_map and set with two custom my_vertex objects.

6.5 Create an empty graph with custom edges and vertices

Say we want to use our own edge class as graph nodes. This is done in multiple steps:

1. Create a custom edge class, called 'my_edge'

2. Install a new property, called 'edge_custom_type'
3. Use the new property in creating a boost::adjacency_list

6.5.1 Creating the custom edge class

In this example, I create a custom edge class. Here I will show the header file of it, as the implementation of it is not important yet.

Algorithm 47 Declaration of my_edge

```
#ifndef MY_EDGE_H
#define MY_EDGE_H

#include <string>

class my_edge
{
public:
  my_edge(
    const std::string& name = "",
    const std::string& description = "",
    const double width = 1.0,
    const double height = 1.0
  ) noexcept;
  std::string m_name;
  std::string m_description;
  double m_width;
  double m_height;
};

bool operator==(const my_edge& lhs, const my_edge& rhs)
    noexcept;

#endif // MY_EDGE_H
```

my_edge is a class that has multiple properties: two doubles 'm_width' ('m_' stands for member) and 'm_height', and two std::strings m_name and m_description. my_edge is copyable, but cannot trivially be converted to a std::string.

6.5.2 Installing the new property

Installing a new property would have been easier, if 'more C++ compilers were standards conformant' ([2], chapter 3.6, footnote at page 52). Boost.Graph uses the BOOST_INSTALL_PROPERTY macro to allow using a custom property:

Algorithm 48 Installing the edge_custom_type property

```
#include <boost/graph/properties.hpp>

namespace boost {
  enum edge_custom_type_t { edge_custom_type = 3142 };
  BOOST_INSTALL_PROPERTY(edge, custom_type);
}
```

The enum value 3142 must be unique.

6.5.3 Create the empty graph with custom edges and vertices

Algorithm 49 Creating an empty graph with custom vertices

```
#include <boost/graph/adjacency_list.hpp>
#include "install_edge_custom_type.h"
#include "install_vertex_custom_type.h"
#include "my_vertex.h"
#include "my_edge.h"

boost::adjacency_list<
  boost::vecS,
  boost::vecS,
  boost::undirectedS,
  boost::property<
    boost::vertex_custom_type_t,my_vertex
  >,
  boost::property<
    boost::edge_custom_type_t,my_edge
  >
>
create_empty_custom_edges_and_vertices_graph() noexcept
{
  return boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    boost::property<
      boost::vertex_custom_type_t,my_vertex
    >,
    boost::property<
      boost::edge_custom_type_t,my_edge
    >
  >();
}
```

This graph:

- has its out edges stored in a std::vector (due to the first boost::vecS)
- has its vertices stored in a std::vector (due to the second boost::vecS)
- is undirected (due to the boost::undirectedS)
- The vertices have one property: they have a custom type, that is of data type my_vertex (due to the boost::property< boost::vertex_custom_type_t, my_vertex>')

- The edges have one property: they have a custom type, that is of data type my_edge (due to the boost::property< boost::edge_custom_type_t, my_edge>')
- The graph has no properties
- Edges are stored in a std::list

The boost::adjacency_list has a new, fifth template argument 'boost::property< boost::edge_custom_type_t, my_edge>'. This can be read as: "edges have the property 'boost::edge_custom_type_t', which is of data type 'my_edge"'. Or simply: "edges have a custom type called my_edge".

6.6 Add a custom edge

Adding a custom edge is very similar to adding a named edge (chapter 4.6).

Algorithm 50 Add a custom edge

```
#include <cassert>
#include <boost/graph/adjacency_list.hpp>
#include "install_edge_custom_type.h"
#include "install_vertex_custom_type.h"
#include "my_vertex.h"
#include "my_edge.h"

template <typename graph>
void add_custom_edge(graph& g, const my_edge& v)
{
  const auto vd_a = boost::add_vertex(g);
  const auto vd_b = boost::add_vertex(g);

  const auto aer = boost::add_edge(vd_a, vd_b, g);
  assert(aer.second);
  const auto my_edge_map = boost::get(boost::
      edge_custom_type,g);
  my_edge_map[aer.first] = v;
}
```

When having added a new (abstract) edge to the graph, the edge descriptor is used to set the my_edge in the graph its my_edge map (using 'boost::get(boost::edge_custom_type,g)').

6.7 Creating K_3 with custom edges and vertices

Instead of using edges with a name, or other properties, here we use a custom edge class called 'my_edge'.

We reproduce the K_3 with named edges and vertices of chapter 4.8 , but with our custom edges and vertices intead:

Algorithm 51 Creating K_3 as depicted in figure 3

```
#include "install_vertex_custom_type.h"
#include "my_vertex.h"
#include "create_empty_custom_edges_and_vertices_graph.h"

#include <boost/graph/adjacency_list.hpp>

boost::adjacency_list<
  boost::vecS,
  boost::vecS,
  boost::undirectedS,
  boost::property<
    boost::vertex_custom_type_t,my_vertex
  >,
  boost::property<
    boost::edge_custom_type_t,my_edge
  >
>
create_custom_edges_and_vertices_k3_graph() noexcept
{
  auto g = create_empty_custom_edges_and_vertices_graph()
     ;
  const auto vd_a = boost::add_vertex(g);
  const auto vd_b = boost::add_vertex(g);
  const auto vd_c = boost::add_vertex(g);
  const auto aer_a = boost::add_edge(vd_a, vd_b, g);
  const auto aer_b = boost::add_edge(vd_b, vd_c, g);
  const auto aer_c = boost::add_edge(vd_c, vd_a, g);
  assert(aer_a.second);
  assert(aer_b.second);
  assert(aer_c.second);

  auto my_vertex_map = boost::get(boost::
     vertex_custom_type,g);
  my_vertex_map[vd_a]
    = my_vertex("top","source",0.0,0.0);
  my_vertex_map[vd_b]
    = my_vertex("right","target",3.14,0);
  my_vertex_map[vd_c]
    = my_vertex("left","target",0,3.14);

  auto my_edge_map = boost::get(boost::edge_custom_type,g
     );
  my_edge_map[aer_a.first]
    = my_edge("AB","first",0.0,0.0);
  my_edge_map[aer_b.first]
    = my_edge("BC","second",3.14,3.14);
  my_edge_map[aer_c.first]
    = my_edge("CA","third",3.14,3.14);

  return g;
}
```

Most of the code is a slight modification of algorithm 39. In the end, the my_edges and my_vertices are obtained as a boost::property_map and set with the custom my_edge and my_vertex objects.

7 Measuring simple graphs traits of a graph with custom edges and vertices

7.1 Count vertex my_vertex

count_vertex_my_vertex

7.2 Find a my_vertex

find_my_vertex

7.3 Find the vertices connected to a certain my_vertex

find_vertices_connected_to_my_vertex

8 Modifying simple graphs traits

It is useful to be able to modify every aspect of a graph. Adding nodes and edges are found in earlier chapters.

8.1 Setting all vertices' names

When the vertices of a graph have named vertices, one set their names as such:

Algorithm 52 Setting the vertices' names

```
#include <string>
#include <vector>

#include <boost/graph/graph_traits.hpp>
#include <boost/graph/properties.hpp>

//TODO: generalize 'names'
template <typename graph>
void set_vertex_names(
  graph& g,
  const std::vector<std::string>& names
)
{
  const auto vertex_name_map = get(boost::vertex_name,g);

  auto names_begin = std::begin(names);
  const auto names_end = std::end(names);
  for (auto vi = vertices(g);
    vi.first != vi.second;
    ++vi.first, ++names_begin)
  {
    assert(names_begin != names_end);
    put(vertex_name_map, *vi.first,*names_begin);
  }
}
```

An impressive feature is that getting the property map holding the graph its names is not a copy, but a reference. Otherwise, modifying 'name_map' (obtained by non-reference) would only modify a copy.

8.2 Setting all vertices' my_vertex objects

When the vertices of a graph are associated with my_vertex objects, one can set these my_vertexes as such:

Algorithm 53 Setting the vertices' my_vertexes

```
#include <string>
#include <vector>

#include <boost/graph/graph_traits.hpp>
#include <boost/graph/properties.hpp>

#include "install_vertex_custom_type.h"
#include "my_vertex.h"

//TODO: generalize 'my_vertexes'
template <typename graph>
void set_vertex_my_vertexes(
  graph& g,
  const std::vector<my_vertex>& my_vertexes
)
{
  const auto my_vertex_map = get(boost::
      vertex_custom_type,g);

  auto my_vertexes_begin = std::begin(my_vertexes);
  const auto my_vertexes_end = std::end(my_vertexes);
  for (auto vi = vertices(g);
    vi.first != vi.second;
    ++vi.first, ++my_vertexes_begin)
  {
    assert(my_vertexes_begin != my_vertexes_end);
    put(my_vertex_map, *vi.first,*my_vertexes_begin);
  }
}
```

An impressive feature is that getting the property map holding the graph its names is not a copy, but a reference. Otherwise, modifying 'my_vertexes_map' (obtained by non-reference) would only modify a copy.

8.3 Replace a vertex its name

rename_vertex

8.4 Replace an edge its name

rename_edge

8.5 Replace a my_vertex

replace_my_vertex

8.6 Clear a named vertex

clear_named_vertex

- void clear_vertex(vertex_descriptor u, adjacency_list& g) . Removes all edges to and from u
- void clear_out_edges(vertex_descriptor u, adjacency_list& g) . Removes all outgoing edges from vertex u in the directed graph g (not applicable for undirected graphs)
- void clear_in_edges(vertex_descriptor u, adjacency_list& g) . Removes all incoming edges from vertex u in the directed graph g (not applicable for undirected graphs)

8.7 Remove a named vertex

remove_named_vertex

8.8 Remove a named edge

remove_named_vertex

- void remove_edge(vertex_descriptor u, vertex_descriptor v, adjacency_list& g) . Removes an edge from g
- void remove_edge(edge_descriptor e, adjacency_list& g) . Removes an edge from g

8.9 Remove a my_vertex

remove_my_vertex

- void remove_vertex(vertex_descriptor u, adjacency_list& g) . Removes a vertex from graph g (It is expected that all edges associated with this vertex have already been removed using clear_vertex or another appropriate function.)

9 Visualizing graphs

Before graphs are visualized, they are stored as a file first. Here, I use the .dot file format.

9.1 Storing a graph as a .dot

Graph are easily saved to a .dot file:

Algorithm 54 Storing a graph as a .dot file

```
#include <fstream>
#include <boost/graph/graphviz.hpp>

///Save a graph to a .dot file
template <typename graph>
void save_graph_to_dot(const graph& g, const std::string&
    filename)
{
  std::ofstream f(filename);
  boost::write_graphviz(f,g);
}
```

Using the create_k2_graph function (algorithm 17) to create a K_2 graph, the .dot file created is displayed in algorithm 55:

Algorithm 55 .dot file created from the create_k2_graph function (algorithm 17)

```
graph G {
0;
1;
0--1 ;
}
```

This .dot file corresponds to figure 4:

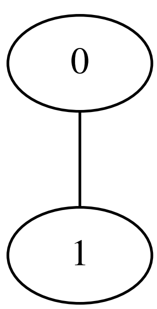

Figure 4: .svg file created from the create_k2_graph function (algorithm 17)

If you used the create_named_vertices_k2_graph function (algorithm 31) to produce a K_2 graph with named vertices, you see that the .dot file does not have stored the vertex names:

Algorithm 56 .dot file created from the create_named_vertices_k2_graph function (algorithm 31)

```
graph G {
0;
1;
0--1 ;
}
```

So, the 'save_graph_to_dot' function (algorithm 54) saves the structure of the graph.

9.2 Storing a graph with named vertices as a .dot

If you used the create_named_vertices_k2_graph function (algorithm 31) to produce a K_2 graph with named vertices, you can store these names additionally with algorithm 57:

Algorithm 57 Storing a graph with named vertices as a .dot file

```
#include <string>
#include <fstream>
#include <boost/graph/graphviz.hpp>
#include <boost/graph/properties.hpp>

#include "get_vertex_names.h"

///Save a graph with named vertices to a .dot file
template <typename graph>
void save_named_vertices_graph_to_dot(const graph& g,
    const std::string& filename)
{
  std::ofstream f(filename);
  const auto names = get_vertex_names(g);
  boost::write_graphviz(f,g,boost::make_label_writer(&
      names[0]));
}
```

The .dot file created is displayed in algorithm 58:

Algorithm 58 .dot file created from the create_named_vertices_k2_graph function (algorithm 31)

```
graph G {
0[label=from];
1[label=to];
0--1 ;
}
```

This .dot file corresponds to figure 5:

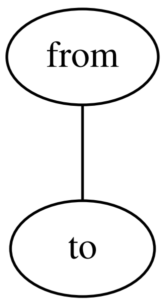

Figure 5: .svg file created from the create_k2_graph function (algorithm 31)

If you used the create_named_edges_and_vertices_k3_graph function (algorithm 39) to produce a K_3 graph with named edges and vertices, you see that the .dot file does not have stored the edge names:

Algorithm 59 .dot file created from the create_named_edges_and_vertices_k3_graph function (algorithm 39)

```
graph G {
0[label=top];
1[label=right];
2[label=left];
0--1 ;
1--2 ;
2--0 ;
}
```

So, the 'save_named_vertices_graph_to_dot' function (algorithm 54) saves only the structure of the graph and its vertex names.

9.3 Storing a graph with named vertices and edges as a .dot

If you used the create_named_edges_and_vertices_k3_graph function (algorithm 39) to produce a K_3 graph with named edges and vertices, you can store these names additionally with algorithm 60:

Algorithm 60 Storing a graph with named edges and vertices as a .dot file

```
#include <string>
#include <fstream>
#include <boost/graph/graphviz.hpp>
#include <boost/graph/properties.hpp>

#include "get_edge_names.h"
#include "get_vertex_names.h"

///Save a graph with named vertices to a .dot file
template <typename graph>
void save_named_edges_and_vertices_graph_to_dot(const
    graph& g, const std::string& filename)
{
  std::ofstream f(filename);
  const auto vertex_names = get_vertex_names(g);
  const auto edge_name_map = boost::get(boost::edge_name,
      g);
  boost::write_graphviz(
    f,
    g,
    boost::make_label_writer(&vertex_names[0]),
    [edge_name_map](std::ostream& out, const auto& e) {
      out << "[label=\"" << edge_name_map[e] << "\"]";
    }
  );
}
```

Note that this algorithm uses C++17.

The .dot file created is displayed in algorithm 61:

Algorithm 61 .dot file created from the create_named_edges_and_vertices_k3_graph function (algorithm 31)

```
graph G {
0[label=top];
1[label=right];
2[label=left];
0--1 [label="AB"];
1--2 [label="BC"];
2--0 [label="CA"];
}
```

This .dot file corresponds to figure 6:

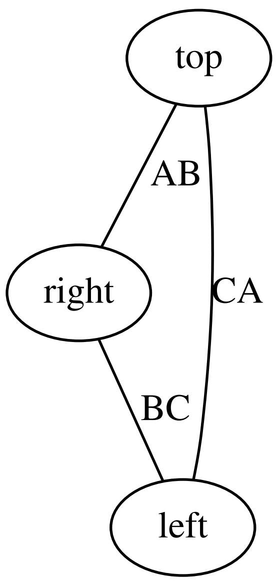

Figure 6: .svg file created from the create_named_edges_and_vertices_k3_graph function (algorithm 31)

If you created a graph with edges more complex than just a name, you will still just write these to the .dot file. Chapter 9.4 shows how to write custom vertices to a .dot file.

So, the 'save_named_edges_and_vertices_graph_to_dot' function (algorithm 54) saves only the structure of the graph and its edge and vertex names.

9.4 Storing a graph with custom vertices as a .dot

If you used the create_custom_vertices_k2_graph function (algorithm 46) to produce a K_2 graph with vertices associated with my_vertex objects, you can store these my_vertexes additionally with algorithm 62:

Algorithm 62 Storing a graph with custom vertices as a .dot file

```
#include <string>
#include <fstream>
#include <boost/graph/graphviz.hpp>
#include <boost/graph/properties.hpp>

#include "get_vertex_my_vertexes.h"

///Save a graph with named vertices to a .dot file
template <typename graph>
void save_custom_vertices_graph_to_dot(const graph& g,
    const std::string& filename)
{
  std::ofstream f(filename);
  const auto my_vertexes = get_vertex_my_vertexes(g);
  boost::write_graphviz(
    f,
    g,
    [my_vertexes](std::ostream& out, const auto& v) {
      const my_vertex m{my_vertexes[v]};
      out << "[label=\""
        << m.m_name
        << ","
        << m.m_description
        << ","
        << m.m_x
        << ","
        << m.m_y
        << "\"]";
    }
  );
}
```

Note that this algorithm uses C++17.

The .dot file created is displayed in algorithm 63:

Algorithm 63 .dot file created from the create_custom_vertices_k2_graph function (algorithm 31)

```
graph G {
0[label="from,source,0,0"];
1[label="to,target,3.14,3.14"];
0--1 ;
}
```

This .dot file corresponds to figure 63:

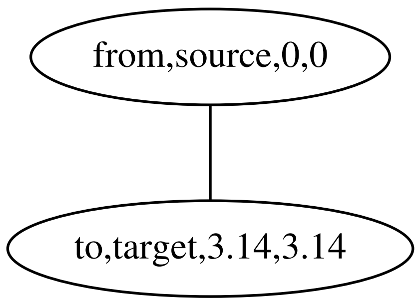

Figure 7: .svg file created from the create_custom_vertices_k2_graph function (algorithm 46)

10 Measuring more complex graphs traits

10.1 Count the number of self-loops

11 Misc functions

11.1 Getting a data type as a std::string

This function will only work under GCC.

Algorithm 64 Getting a data type its name as a std::string

```
#include <string>
#include <typeinfo>
#include <cstdlib>
#include <cxxabi.h>

//From http://stackoverflow.com/questions/1055452/c-get-
    name-of-type-in-template
//Thanks to m-dudley ( http://stackoverflow.com/users
    /111327/m-dudley )
template<typename T>
std::string get_type_name()
{
  std::string tname = typeid(T).name();
  int status = -1;
  char * const demangled_name{
    abi::__cxa_demangle(tname.c_str(), NULL, NULL, &
        status)
  };
  if(status == 0) {
    tname = demangled_name;
    std::free(demangled_name);
  }
  return tname;
}
```

12 Errors

Some common errors.

12.1 Formed reference to void

This compile-time error occurs when you create a graph without a certain property, then subsequently reading that property, as in algorithm 65:

Algorithm 65 Creating the error 'formed reference to void'

```
#include "create_k2_graph.h"
#include "get_vertex_names.h"

void formed_reference_to_void() noexcept
{
  get_vertex_names(create_k2_graph());
}
```

In algorithm 65 a graph is created with vertices of no properties. Then the names of these vertices, which do not exists, are tried to be read. If you want to read the names of the vertices, supply a graph that has this property.

References

[1] John Lakos. *Large-scale C++ software design*, volume 10. Addison-Wesley Reading, 1996.

[2] Jeremy G Siek, Lie-Quan Lee, and Andrew Lumsdaine. *Boost Graph Library: User Guide and Reference Manual, The*. Pearson Education, 2001.

[3] Bjarne Stroustrup. *The C++ Programming Language (3rd edition)*. 1997.

Index