Boost.Graph tutorial

Richel Bilderbeek

December 6, 2015

Contents

1 Introduction 2

1.1 Coding style used 2

1.2 Pitfalls 2

2 Building graphs 3

2.1 Creating an empty graph 3

2.2 Add a vertex 3

2.3 Add an edge 4

2.4 Creating K_2, a fully connected graph with two vertices 5

2.5 Creating an empty graph with named vertices 7

2.6 Add a vertex with a name 8

2.7 Creating K_2 with named vertices 8

2.8 Creating an empty graph with named edges and vertices 11

2.9 Add an edge with a name 12

2.10 Creating K_3 with named edges and vertices 13

2.11 Create an empty graph with custom vertices 16

2.11.1 Creating the custom vertex class 16

2.11.2 Installing the new property 17

2.11.3 Create the empty graph with custom vertices 18

2.12 Add a custom vertex 19

2.13 Creating K_2 with custom vertices 20

2.14 Create an empty graph with custom edges and vertices 22

2.15 Add a custom edge 22

2.16 Creating K_2 with custom edges and vertices 22

3 Measuring simple graphs traits 22

3.1 Getting the vertices 22

3.2 Getting the edges 22

3.3 Counting the number of vertices 22

3.4 Counting the number of edges 23

3.5 Getting the vertices' names 23

3.6 Getting the edges' names 24

3.7 Find a vertex by its name 26

3.8 Replace a vertex its name 26

4 Modifying simple graphs traits **26**

4.1 Setting all vertices' names 26

5 Visualizing graphs **27**

5.1 Storing a graph as a .dot 27

5.2 Storing a graph with named vertices as a .dot 28

5.3 Storing a graph with named vertices and edges as a .dot 30

6 TODO's in general **33**

6.1 Personal experciences with Boost.Graph 33

6.2 To add 33

7 Graph of this tutorial **34**

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 [1] and the Boost.Graph website, both did not satisfy these requirements.

1.1 Coding style used

I use the coding style from the Core C++ Guidelines.

I prefer not to use the keyword auto, but to explicitly mention the type instead. I think this is beneficial to beginners. When using Boost.Graph in production code, I do prefer to use auto.

OTOH, while writing this tutorial, I use auto when I loose too much time figuring out the type

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

1.2 Pitfalls

The choice between 'boost::get', 'std::get' and 'get'. AFAIKS, when in doubt, use 'get'.

2 Building graphs

Boost.Graph is about creating graphs. In this chapter we create graphs, starting from simple to more complex.

2.1 Creating an empty graph

Let's create a trivial empty graph:

Algorithm 1 Creating an empty graph

```
#include "create_empty_graph.h"

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

Congratulations, you've just created a boost::adjacency_list in which:

- The out edges are stored in a std::vector
- The vertices are stored in a std::vector
- The graph is directed
- Vertices, edges and graph have no properties
- 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 Add a vertex

To add a vertex to a graph, the boost::add_vertex function is used as such:

Algorithm 2 Adding a vertex to a graph

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

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

Note that boost::add_vertex returns a vertex descriptor, which is ignored for now. A vertex descriptor can be used to, for example, connect two vertices by an edge

2.3 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. Algorithm 3 adds two vertices to a graph, and connects these two using boost::add_edge:

Algorithm 3 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)
{
  using boost::graph_traits;
  using vertex_descriptor
    = typename graph_traits<graph>::vertex_descriptor;
  using edge_descriptor
    = typename graph_traits<graph>::edge_descriptor;
  using edge_insertion_result
    = std::pair<edge_descriptor,bool>;

  const vertex_descriptor va = boost::add_vertex(g);
  const vertex_descriptor vb = boost::add_vertex(g);
  const edge_insertion_result ea
    = boost::add_edge(va, vb, g);

  assert(ea.second);
}
```

Note that half of the code consists out of using statements. 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_edge returns a std::pair, consisting of an edge descriptor and a boolean success indicator. In algorithm 3 we assert that this insertion was successfull. Insertion can fail if an edge is already present and duplicates are not allowed.

2.4 Creating K_2, a fully connected graph with two vertices

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.

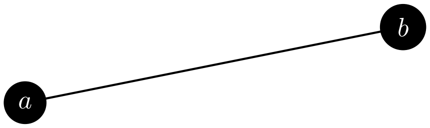

Figure 1: K_2: a fully connected graph with two vertices named a and b

To create K_2, the following code can be used:

Algorithm 4 Creating K_2 as depicted in figure 1

```
#include "create_k2_graph.h"

boost::adjacency_list<
  boost::vecS,
  boost::vecS,
  boost::undirectedS
>
create_k2_graph() noexcept
{
  using boost::graph_traits;

  using graph = boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS
  >;

  using vertex_descriptor
    = typename graph_traits<graph>::vertex_descriptor;
  using edge_descriptor
    = typename graph_traits<graph>::edge_descriptor;
  using edge_insertion_result
    = std::pair<edge_descriptor,bool>;

  graph g;
  const vertex_descriptor va = boost::add_vertex(g);
  const vertex_descriptor vb = boost::add_vertex(g);
  const edge_insertion_result ea
    = boost::add_edge(va, vb, g);
  assert(ea.second);
  return g;
}
```

Note that this code has more lines of using statements than actual code! In this code, the third template argument of boost::adjacency_list is boost::undirectedS, to select (that is what the S means) for an undirected graph. Adding a vertex with boost::add_vertex results in a vertex descriptor, which is a handle to the vertex added to the graph. Two vertex descriptors are then used to add an edge to the graph. Adding an edge using boost::add_edge returns two things: an edge descriptor and a boolean indicating success. In the code example, we assume insertion is successfull.

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

2.5 Creating an empty graph with named vertices

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

Algorithm 5 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
{
  using graph = boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    boost::property<
      boost::vertex_name_t,std::string
    >
  >;

  graph g;
  return g;
}
```

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".

2.6 Add a vertex with a name

Adding a vertex without a name was trivially easy (see chapter 2). Adding a vertex with a name is less easy:

Algorithm 6 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)
{
  using boost::graph_traits;
  using boost::property_map;
  using boost::vertex_name_t;
  using vertex_descriptor
    = typename boost::graph_traits<graph>::
        vertex_descriptor;
  using name_map_t
    = typename property_map<graph,vertex_name_t>::type;

  const vertex_descriptor vd{
    boost::add_vertex(g)
  };

  name_map_t name_map{boost::get(boost::vertex_name,g)};
  name_map[vd] = name;
}
```

Instead of calling 'boost::add_vertex' with an additional argument containing the name of the vertex, multiple things need to be done. When adding a new vertex to the graph, the vertex descriptor 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.

2.7 Creating K_2 with named vertices

We extend K_2 of chapter 2.4 by naming the vertices 'from' and 'to', as depicted in figure 2:

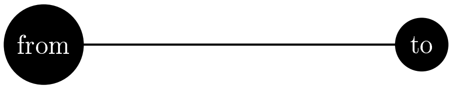

Figure 2: K_2: a fully connected graph with two vertices with the text 'from' and 'to'

To create K_2, the following code can be used:

Algorithm 7 Creating K_2 as depicted in figure 2

```
#include "create_named_vertices_k2_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
{
  using graph = boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    boost::property<
      boost::vertex_name_t,std::string
    >
  >;
  using vertex_descriptor
    = typename boost::graph_traits<graph>::
        vertex_descriptor;
  using edge_descriptor
    = typename boost::graph_traits<graph>::
        edge_descriptor;
  using edge_insertion_result
    = std::pair<edge_descriptor,bool>;
  using name_map_t
    = boost::property_map<graph,boost::vertex_name_t>::
        type;

  graph g;
  const vertex_descriptor va = boost::add_vertex(g);
  const vertex_descriptor vb = boost::add_vertex(g);
  const edge_insertion_result ea
    = boost::add_edge(va, vb, g);
  assert(ea.second);

  //Add names
  name_map_t name_map{boost::get(boost::vertex_name,g)};
  name_map[va] = "from";
  name_map[vb] = "to";

  return g;
}
```

Most of the code is a repeat of algorithm 4. In the end, the names are obtained as a boost::property_map and set.

2.8 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 8 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
{
  using graph = 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
    >
  >;

  graph g;
  return g;
}
```

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".

2.9 Add an edge with a name

Adding an edge with a name:

Algorithm 9 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)
{
  using boost::edge_name_t;
  using boost::graph_traits;
  using boost::property_map;
  using vertex_descriptor
    = typename graph_traits<graph>::vertex_descriptor;
  using edge_descriptor
    = typename graph_traits<graph>::edge_descriptor;
  using edge_insertion_result
    = std::pair<edge_descriptor,bool>;
  using name_map_t
    = typename property_map<graph,edge_name_t>::type;

  const vertex_descriptor va = boost::add_vertex(g);
  const vertex_descriptor vb = boost::add_vertex(g);
  const edge_insertion_result ea
    = boost::add_edge(va, vb, g);

  assert(ea.second);

  name_map_t name_map{
    boost::get(boost::edge_name,g)
  };
  name_map[ea.first] = edge_name;

}
```

In this code snippet, the edge descriptor when using 'boost::add_edge' is used as a key to change the edge name map.

2.10 Creating K_3 with named edges and vertices

We extend the graph K_2 with named vertices of chapter 2.7 by adding names to the edges, as depicted in figure 3:

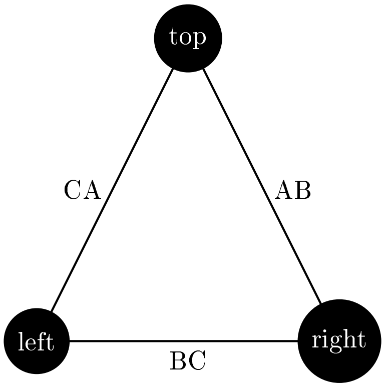

Figure 3: K_3: a fully connected graph with three named edges and vertices

To create K_3, the following code can be used:

Algorithm 10 Creating K_3 as depicted in figure 3

```
#include "create_named_edges_and_vertices_k3_graph.h"

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
{
  using graph = 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
    >
  >;
  using vertex_descriptor
    = typename boost::graph_traits<graph>::
        vertex_descriptor;
  using edge_descriptor
    = typename boost::graph_traits<graph>::
        edge_descriptor;
  using edge_insertion_result
    = std::pair<edge_descriptor,bool>;
  using vertex_name_map_t
    = boost::property_map<graph,boost::vertex_name_t>::
        type;
  using edge_name_map_t
    = boost::property_map<graph,boost::edge_name_t>::type
        ;

  graph g;
  const vertex_descriptor va = boost::add_vertex(g);
  const vertex_descriptor vb = boost::add_vertex(g);
  const vertex_descriptor vc = boost::add_vertex(g);
  const edge_insertion_result eab
    = boost::add_edge(va, vb, g);
  assert(eab.second);
  const edge_insertion_result ebc
    = boost::add_edge(vb, vc, g);
  assert(ebc.second);
  const edge_insertion_result eca
    = boost::add_edge(vc, va, g);
  assert(eca.second);

  //Add vertex names
  vertex_name_map_t vertex_name_map{boost::get(boost::
      vertex_name,g)};
```

Most of the code is a repeat of algorithm 7. In the end, the edge names are obtained as a boost::property_map and set.

2.11 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

2.11.1 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 11 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
  );
  std::string m_name;
  std::string m_description;
  double m_x;
  double m_y;
};

#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.

2.11.2 Installing the new property

Installing a new property would have been easier, if 'more C++ compilers were standards conformant' ([1], chapter 3.6, footnote at page 52). Boost.Graph uses the BOOST_INSTALL_PROPERTY macro to allow using a custom property:

Algorithm 12 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.

2.11.3 Create the empty graph with custom vertices

Algorithm 13 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
{
  using graph = boost::adjacency_list<
  boost::vecS,
  boost::vecS,
  boost::undirectedS,
  boost::property<
    boost::vertex_custom_type_t,my_vertex
    >
  >;

  graph g;
  return g;
}
```

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”.

2.12 Add a custom vertex

Adding a custom vertex is very similar to adding a named vertex (chapter 6).

Algorithm 14 Add a vertex with a name

```
#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)
{
  using boost::graph_traits;
  using boost::property_map;
  using boost::vertex_name_t;
  using boost::vertex_custom_type_t;
  using vertex_descriptor
    = typename boost::graph_traits<graph>::
        vertex_descriptor;
  using my_vertex_map_t
    = typename property_map<graph,vertex_custom_type_t>::
        type;

  const vertex_descriptor vd{
    boost::add_vertex(g)
  };

  my_vertex_map_t name_map{boost::get(boost::
      vertex_custom_type,g)};
  name_map[vd] = 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)').

2.13 Creating K_2 with custom vertices

We reproduce the K_2 with named vertices of chapter 2.7 , but with our custom vertices intead:

Algorithm 15 Creating K_2 as depicted in figure 2

```
#include "install_vertex_custom_type.h"
#include "my_vertex.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
{
  using boost::property_map;
  using boost::vertex_custom_type_t;
  using graph = boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    boost::property<
      boost::vertex_custom_type_t,my_vertex
    >
  >;
  using vertex_descriptor
    = typename boost::graph_traits<graph>::
        vertex_descriptor;
  using edge_descriptor
    = typename boost::graph_traits<graph>::
        edge_descriptor;
  using edge_insertion_result
    = std::pair<edge_descriptor,bool>;
  using my_vertex_map_t
    = typename property_map<graph,vertex_custom_type_t>::
        type;

  graph g;
  const vertex_descriptor va = boost::add_vertex(g);
  const vertex_descriptor vb = boost::add_vertex(g);
  const edge_insertion_result ea
    = boost::add_edge(va, vb, g);
  assert(ea.second);

  //Add names
  my_vertex_map_t name_map{boost::get(boost::
     vertex_custom_type,g)};
  name_map[va] = my_vertex("from");
  name_map[vb] = my_vertex("to");

  return g;
}
```

Most of the code is a slight modification of algorithm 7. In the end, the my_vertices are obtained as a boost::property_map and set with two custom my_vertex objects.

2.14 Create an empty graph with custom edges and vertices

2.15 Add a custom edge

2.16 Creating K_2 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'.

3 Measuring simple graphs traits

Measuring simple traits of the graphs created allows you to debug your code.

3.1 Getting the vertices

You can use boost::vertices to obtain an iterator pair. The first iterator points to the first vertex, the second points to beyond the last vertex.

3.2 Getting the edges

You can use boost::edges to obtain an iterator pair. The first iterator points to the first edge, the second points to beyond the last edge.

3.3 Counting the number of vertices

Use boost::num_vertices, as shown here:

Algorithm 16 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));
}
```

3.4 Counting the number of edges

Use boost::num_edges, as shown here:

Algorithm 17 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));
}
```

3.5 Getting the vertices' names

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

Algorithm 18 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)
{
  using vertex_iterator
    = typename boost::graph_traits<graph>::
        vertex_iterator;
  using vertex_iterators
    = std::pair<vertex_iterator,vertex_iterator>;

  std::vector<std::string> v;

  //TODO: remove auto
  const auto name_map = get(boost::vertex_name,g);

  for (vertex_iterators p = vertices(g);
    p.first != p.second;
    ++p.first)
  {
    v.emplace_back(get(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.

When trying to get the vertices' names from a graph without vertices with names, you will get the error 'formed reference to void' (for example, with the code 'get_vertex_names(create_k2_graph());').

3.6 Getting the edges' names

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

Algorithm 19 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)
{
  using boost::graph_traits;
  using edge_iterator
    = typename graph_traits<graph>::edge_iterator;
  using edge_iterators
    = std::pair<edge_iterator,edge_iterator>;

  std::vector<std::string> v;

  //TODO: remove auto
  const auto name_map = get(boost::edge_name,g);

  for (edge_iterators p = boost::edges(g);
    p.first != p.second;
    ++p.first)
  {
    v.emplace_back(get(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.

When trying to get the edges' names from a graph without vertices with names, you will get the error 'formed reference to void' (for example, with the code 'get_vertex_names(create_k2_graph());').

3.7 Find a vertex by its name

3.8 Replace a vertex its name

4 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.

4.1 Setting all vertices' names

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

Algorithm 20 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
)
{
  using vertex_iterator
    = typename boost::graph_traits<graph>::
        vertex_iterator;
  using vertex_iterators
    = std::pair<vertex_iterator,vertex_iterator>;

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

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

5 Visualizing graphs

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

5.1 Storing a graph as a .dot

Graph are easily saved to a .dot file:

Algorithm 21 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 4) to create a K_2 graph, the .dot file created is displayed in algorithm 22:

Algorithm 22 .dot file created from the create_k2_graph function (algorithm 4)

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

This .dot file corresponds to figure 4:

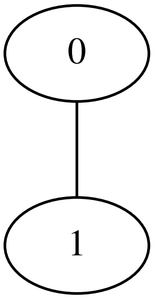

Figure 4: .svg file created from the create_k2_graph function (algorithm 4)

If you used the create_named_vertices_k2_graph function (algorithm 7) to produce a K_2 graph with named vertices, you see that the .dot file does not have stored the vertex names:

Algorithm 23 .dot file created from the create_named_vertices_k2_graph function (algorithm 7)

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

So, the 'save_graph_to_dot' function (algorithm 21) saves the structure of the graph.

5.2 Storing a graph with named vertices as a .dot

If you used the create_named_vertices_k2_graph function (algorithm 7) to produce a K_2 graph with named vertices, you can store these names additionally with algorithm 24:

Algorithm 24 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 25:

Algorithm 25 .dot file created from the create_named_vertices_k2_graph function (algorithm 7)

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

This .dot file corresponds to figure 5:

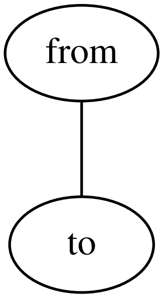

Figure 5: .svg file created from the create_k2_graph function (algorithm 7)

If you used the create_named_edges_and_vertices_k3_graph function (algorithm 10) 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 26 .dot file created from the create_named_edges_and_vertices_k3_graph function (algorithm 10)

```
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 21) saves only the structure of the graph and its vertex names.

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

If you used the create_named_edges_and_vertices_k3_graph function (algorithm 10) to produce a K_3 graph with named edges and vertices, you can store these names additionally with algorithm 27:

Algorithm 27 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] << "\"]";
    }
  );
}
```

The .dot file created is displayed in algorithm 28:

Algorithm 28 .dot file created from the create_named_edges_and_vertices_k3_graph function (algorithm 7)

```
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:

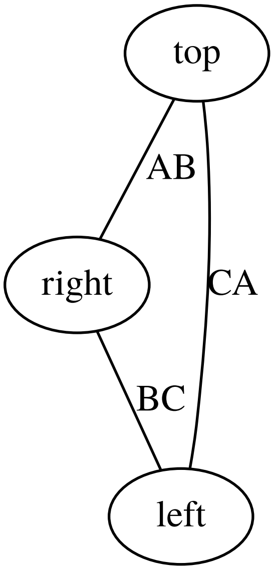

Figure 6: .svg file created from the create_named_edges_and_vertices_k3_graph function (algorithm 7)

If you used the MORE_COMPLEX_create_named_edges_and_vertices_k3_graph function (algorithm 10) 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 29 .dot file created from the MORE_COMPLEX_create_named_edges_and_vertices_k3_graph function (algorithm 10)

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

So, the 'save_named_edges_and_vertices_graph_to_dot' function (algorithm 21) saves only the structure of the graph and its edge and vertex names.

6 TODO's in general

6.1 Personal experciences with Boost.Graph

I have been experimenting with Boost.Graph since 2006 and I both use the documentation on the Boost website [1] and the book [1].

- Boost.Graph seems like the most type-safe extendable graph library around: learning it will pay off!
- Boost.Graph requires a good compiler, like GCC
- The book [2] will not help a beginner: all code snippets are written 'too smart', where a beginner might prefer four easy-to-understand lines, instead of one using magic. Most code snippets are scattered as part of a complete project, where a beginner might prefer simple short projects
- The website [1] will not help a beginner, for the same reasons as above
- I have never found Boost.Graph documentation or code snippets a beginner would understand (except for (hopefully) at my website), as if nobody understands Boost.Graph and/or everybody that understands Boost.Graph cannot explain simply anymore
- I learned most of Boost.Graph from my IDE (the helpful Qt Creator): by viewing the code that failed, I could understand what was expected for me. I would never have understood Boost.Graph if I would have used a text editor for programming

6.2 To add

Code snippets show include guards , use header_impl instead.

- std::pair<edge_iterator, edge_iterator> edges(const adjacency_list& g) . Returns an iterator pair corresponding to the edges in graph g
- vertex_descriptor source(edge_descriptor e, const adjacency_list& g) . Returns the source vertex of an edge
- vertex_descriptor target(edge_descriptor e, const adjacency_list& g) . Returns the target vertex of an edge
- degree_size_type in_degree(vertex_descriptor u, const adjacency_list& g) . Returns the in-degree of a vertex
- degree_size_type out_degree(vertex_descriptor u, const adjacency_list& g) . Returns the out-degree of a 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
- 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)
- 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.)

7 Graph of this tutorial

This tutorial would not be complete with a graph that connects all chapter chronologically:

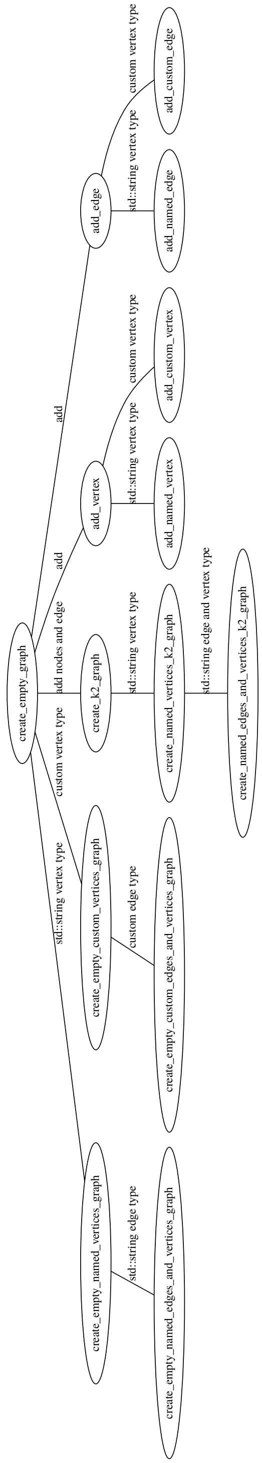

Figure 7: .svg file created from the create_tutorial_chapters_graph function

References

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

Index

K_2 with named vertices, create, 8
K_2, create, 5
K_3 with named edges and vertices, create, 13
100π, 17

Add a vertex, 3
Add an edge, 4
Add named edge, 12
Add named vertex, 8
add_edge, 4
add_named_edge, 13
add_named_vertex, 8, 19
add_vertex, 3
assert, 4

boost::add_edge, 4, 6, 13
boost::add_vertex, 3, 6
boost::adjacency_list, 3, 6, 7, 12, 19
boost::adjacency_matrix, 3
boost::edge_name_t, 12
boost::get(boost::vertex_custom_type,g), 19
boost::get(boost::vertex_name,g), 8
boost::num_edges, 23
boost::num_vertices, 22
boost::property, 7, 12, 18
boost::property<boost::edge_name_t,std::string>, 12
boost::property<boost::vertex_custom_type_t,my_vertex>, 18, 19
boost::property<boost::vertex_name_t,std::string>, 7, 12
boost::undirectedS, 6, 7, 11, 18
boost::vecS, 7, 11, 18
boost::vertex_custom_type, 19
boost::vertex_custom_type_t, 18, 19
boost::vertex_name, 8
boost::vertex_name_t, 7, 12
BOOST_INSTALL_PROPERTY, 17

Counting the number of edges, 23
Counting the number of vertices, 22
Create K_2, 5
Create K_2 with named vertices, 8
Create K_3 with named edges and vertices, 13
Create .dot from graph, 27
Create .dot from graph with named edges and vertices, 30
Create .dot from graph with named vertices, 28
Create an empty graph, 3
Create an empty graph with named edges and vertices, 11
Create an empty graph with named vertices, 7
create_custom_vertices_k2_graph, 21
create_empty_custom_vertices_graph, 18
create_empty_graph, 3
create_empty_named_edges_and_vertices_graph, 11
create_empty_named_vertices_graph, 7
create_k2_graph, 6
create_named_edges_and_vertices_k3, 15
create_named_vertices_k2_graph, 10
create_tutorial_chapters_graph, 35

Declaration, my_vertex, 16

edge descriptor, 13
Edge, add, 4
edges, counting, 23
Empty graph with named edges and vertices, create, 11
Empty graph with named vertices, create, 7
Empty graph, create, 3

get_edge_names, 25
get_n_edges, 23
get_n_vertices, 22
get_vertex_names, 24

install_vertex_custom_type, 17

m_, 16
macro, 17
member, 16
my_vertex, 16
my_vertex declaration, 16
my_vertex.h, 16

Named edge, add, 12
Named edges and vertices, create empty graph, 11
Named vertex, add, 8
Named vertices, create empty graph, 7

Save graph as .dot, 27
Save graph with name edges and vertices as .dot, 30
Save graph with name vertices as .dot, 28
save_graph_to_dot, 27
save_named_edges_and_vertices_graph_to_dot, 31
save_named_vertices_graph_to_dot, 29
Set vertex names, 26
set_vertex_names, 26
std::pair, 4

vertex descriptor, 4, 6
Vertex, add, 3
Vertex, add named, 8
vertex_custom_type, 16
vertices, counting, 22
Vertices, set names, 26