Boost.Graph tutorial

Richel Bilderbeek

December 5, 2015

1 Introduction

I think that Boost.Graph is designed very well. Drawback is IMHO that there are only few and even fewer complete examples using Boost.Graph.

The book [1] is IMHO not suited best for a tutorial as it contains heavy templated code, and an unchronological ordering of subjects. More experienced programmers can appreciate that the authors took great care that the code snippets written in the book were correct: all snippets are numbered, and I'd bet they are tested to compile.

1.1 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

1.2 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.

1.3 Pitfalls

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

1.4 TODO's in general

Code snippets show include guards , use header_imp instead.

- std::pair<edge_iterator, edge_iterator> edges(const adjacency_list& g) . Returns an iterator pair corresponding to the edges in graph g
- vertices_size_type num_vertices(const adjacency_list& g) . Returns the number of vertices in g
- edges_size_type num_edges(const adjacency_list& g) . Returns the number of edges in 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.)

2 Creating 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 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.

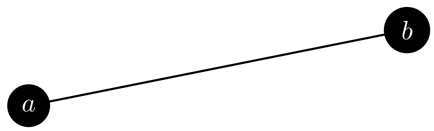

Figure 1: K_2: a fully connected graph with two vertices named a and b

To create K_2, the following code can be used:

Algorithm 2 Creating K_2 as depicted in figure1

```
#include "create_k2_graph.h"

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

  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.

2.3 Creating K_2 with named vertices

We extend K_2 of chapter 2.2 by naming the vertices 'from' and 'to', as depicted in figure 2:

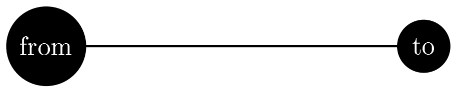

Figure 2: K_2: a fully connected graph with two vertices with the text 'from' and 'to'

To create K_2, the following code can be used:

Algorithm 3 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 2. In the end, the names are obtained as a boost::property_map and set.

2.4 Creating K_3 with named edges and vertices

We extend the graph K_2 with named vertices of chapter 2.3 by adding names to the edges, as depicted in figure 3:

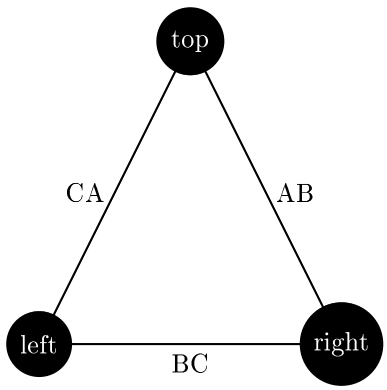

Figure 3: K_3: a fully connected graph with three named edges and vertices

To create K_3, the following code can be used:

Algorithm 4 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 3. In the end, the edge names are obtained as a boost::property_map and set.

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 5 Count the numbe of vertices

```
#ifndef GET_N_VERTICES_H
#define GET_N_VERTICES_H

#include <utility>
#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));
  /*
  using vertex_iterator
    = typename boost::graph_traits<graph>::
        vertex_iterator;
  using vertex_iterators
    = std::pair<vertex_iterator,vertex_iterator>;

  const vertex_iterators vertex_iters
    = boost::vertices(g);
  return std::distance(
    vertex_iters.first,
    vertex_iters.second
  );
  */
}

#endif // GET_N_VERTICES_H
```

3.4 Counting the number of edges

Use boost::num_edges, as show here:

Algorithm 6 Count the number of edges

```
#ifndef GET_N_EDGES_H
#define GET_N_EDGES_H

#include <utility>
#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));
  /*
  const auto edge_iters = boost::edges(g);
  return std::distance(edge_iters.first,edge_iters.second
      );
  */
}

#endif // GET_N_EDGES_H
```

3.5 Getting the vertices' names

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

Algorithm 7 Get the vertices' names

```
#ifndef GET_VERTEX_NAMES_H
#define GET_VERTEX_NAMES_H

#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 = boost::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;
}

//TODO: generalize to return any type
template <typename graph>
std::vector<std::string> get_vertex_names_DOESNOTWORK(
    const graph& g)
{
  using vertex_iterator
    = typename boost::graph_traits<graph>::
        vertex_iterator;
  using vertex_iterators
    = std::pair<vertex_iterator,vertex_iterator>;
  using name_map_t
    = typename boost::property_map<graph,boost::
        vertex_name_t>::type;

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

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

  for (vertex_iterators p = vertices(g);
    p.first != p.second;
    ++p.first)
  {
```

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 8 Get the edges' names

```
#ifndef GET_EDGE_NAMES
#define GET_EDGE_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_edge_names(const graph& g)
{
  using edge_iterator
    = typename boost::graph_traits<graph>::edge_iterator;
  using edge_iterators
    = std::pair<edge_iterator,edge_iterator>;

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

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

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

#endif // GET_EDGE_NAMES
```

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.

4.1 Add a vertex

4.2 Add a node

4.3 Setting all vertices' names

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

Algorithm 9 Setting the vertices' names

```
#ifndef SET_VERTEX_NAMES_H
#define SET_VERTEX_NAMES_H


#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);
  }
}


#endif // SET_VERTEX_NAMES_H
```

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 10 Storing a graph as a .dot file

```
#ifndef SAVE_GRAPH_TO_DOT_H
#define SAVE_GRAPH_TO_DOT_H

#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);
}

#endif // SAVE_GRAPH_TO_DOT_H
```

Using the create_k2_graph function (algorithm 2) to create a K_2 graph, the .dot file created is displayed in algorithm 11:

Algorithm 11 .dot file created from the create_k2_graph function (algorithm 2)

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

This .dot file corresponds to figure 4:

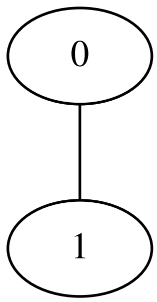

Figure 4: .svg file created from the create_k2_graph function (algorithm 2)

If you used the create_named_vertices_k2_graph function (algorithm 3) to produce a K_2 graph with named vertices, you see that the .dot file does not have stored the vertex names:

Algorithm 12 .dot file created from the create_named_vertices_k2_graph function (algorithm 3)

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

So, the 'save_graph_to_dot' function (algorithm 10) 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 3) to produce a K_2 graph with named vertices, you can store these names additionally with algorithm 13:

Algorithm 13 Storing a graph with named vertices as a .dot file

```
#ifndef SAVE_NAMED_VERTICES_GRAPH_TO_DOT_H
#define SAVE_NAMED_VERTICES_GRAPH_TO_DOT_H

#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]));
}


#endif // SAVE_NAMED_VERTICES_GRAPH_TO_DOT_H
```

The .dot file created is displayed in algorithm 14:

Algorithm 14 .dot file created from the create_named_vertices_k2_graph function (algorithm 3)

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

This .dot file corresponds to figure 5:

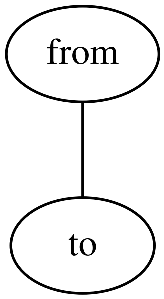

Figure 5: .svg file created from the create_k2_graph function (algorithm 3)

If you used the create_named_edges_and_vertices_k3_graph function (algorithm 4) 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 15 .dot file created from the create_named_edges_and_vertices_k3_graph function (algorithm 4)

```
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 10) 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 4) to produce a K_3 graph with named edges and vertices, you can store these names additionally with algorithm 16:

Algorithm 16 Storing a graph with named edges and vertices as a .dot file

```
#ifndef SAVE_NAMED_EDGES_AND_VERTICES_GRAPH_TO_DOT
#define SAVE_NAMED_EDGES_AND_VERTICES_GRAPH_TO_DOT

#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] << "]";
    }
  );
}

#endif // SAVE_NAMED_EDGES_AND_VERTICES_GRAPH_TO_DOT
```

The .dot file created is displayed in algorithm 17:

Algorithm 17 .dot file created from the create_named_edges_and_vertices_k3_graph function (algorithm 3)

```
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:

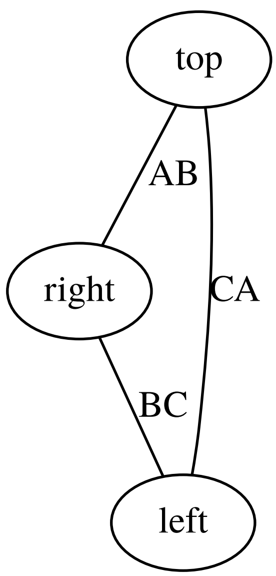

Figure 6: .svg file created from the create_named_edges_and_vertices_k3_graph function (algorithm 3)

If you used the MORE_COMPLEX_create_named_edges_and_vertices_k3_graph function (algorithm 4) 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 18 .dot file created from the MORE_COMPLEX_create_named_edges_and_vertices_k3_graph function (algorithm 4)

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

References

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