Boost.Graph tutorial

Richel Bilderbeek

December 4, 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 Coding style used

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.

1.2 Pitfalls

Do not use 'boost::get', use 'get'

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.

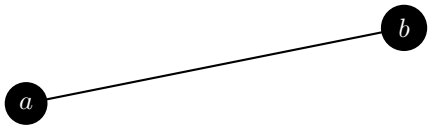

Figure 1: K_2: a fully connected graph with two vertices named a and b

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:

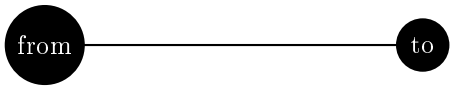

Figure 2: K_2: a fully connected graph with two vertices with the text 'from' and 'to'

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:

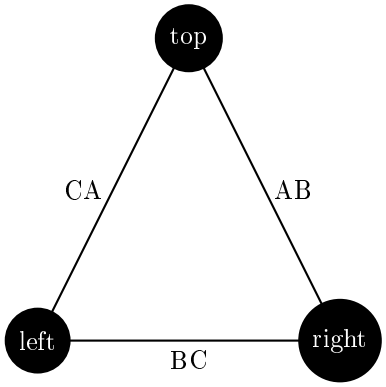

Figure 3: K_3: a fully connected graph with three named edges and vertices

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 Counting the number of vertices

There is na member function to directly obtain the number of vertices. One solution is to use the following code:

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)
{
  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
```

The free function boost::vertices returns a pair of iterators. The first points to the first vertex, the second beyond the last vertex. The STL function std::distance can be used to measure the amount of elements in between. Note that if the vertices are stored in a std::list, this function takes O(n) complexity.

3.2 Counting the number of edges

There is na member function to directly obtain the number of edges. One solution is to use the following code:

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)
{
  const auto edge_iters = boost::edges(g);
  return std::distance(edge_iters.first,edge_iters.second
      );
}

#endif // GET_N_EDGES_H
```

The free function boost::edges returns a pair of iterators. The first points to the first vertex, the second beyond the last vertex. The STL function std::distance can be used to measure the amount of elements in between. Note that if the edges are stored in a std::list, this function takes O(n) complexity.

3.3 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());').

3.4 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());').

3.5 Find a vertex by 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

wef

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

References

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