A well-connected C++14 Boost.Graph tutorial

Richel Bilderbeek

December 14, 2015

Contents

1 Introduction

I needed this tutorial already 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 [8] and the Boost.Graph website, both did not satisfy these requirements.

This tutorial is intended to take the reader to the level of understanding the book [8] and the Boost.Graph website require.

The chapters of this tutorial are also like a well-connected graph. To allow for quicker learners to skim chapters, or for beginners looking to find the patterns, some chapters are repetitions of each other (for example, getting an edge its name is very similar to getting a vertex its name)[1]. This tutorial is not about being short, but being complete[2].

A pivotal chapter is chapter 4.2, 'Finding the first vertex with a name', as this opens up the door to finding a vertex and manipulating it.

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 single 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. The code, as well as this tutorial, can be downloaded from the GitHub at `www.github.com/richelbilderbeek/BoostGraphTutorial`.

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 9.1). 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'. Note that the heavily-use 'get' function must reside in the namespace of the graph to work on. In this tutorial, this is in the global namespace. Thus, I will write 'get', instead of 'boost::get', as the latter does not compile.

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.

[1]There was even copy-pasting involved!

[2]at the risk of being called bloated

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 2.14

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

- Counting the number of vertices: see chapter 2.3
- Counting the number of edges: see chapter 2.4
- Adding a vertex: see chapter 2.5
- Getting all vertices: see chapter 2.7
- Getting all vertex descriptors: see chapter 2.8
- Adding an edge: see chapter 2.9
- Getting all edges: see chapter 2.11
- Getting all edge descriptors: see chapter 2.13
- Getting the vertices' out degrees: see chapter 2.16

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

The chapter also introduces some important concepts:

- Vertex descriptors: see chapter 2.6
- Edge insertion result: see chapter 2.10
- Edge descriptors: see chapter 2.12

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<>();
}
```

The code consists out of an #include and a function definition. The #include includes the header file for the boost::adjacency_list class. Without including this file, you will get compile errors like 'definition of boost::adjacency_list unknown'[3]. The function 'create_empty_directed_graph' has:

- a return type: The return type is 'boost::adjacency_list<>', that is a 'boost::adjacency_list with all template arguments set at their defaults
- a noexcept specification: the function should not throw[4], so it is preferred to mark it noexcept[5].
- a function body: all the function body does is create a 'boost::adjacency_list<>' by calling its constructor, by using the round brackets

Algorithm 2 demonstrates the 'create_empty_directed_graph' function. Auto is used, as this is preferred over explicit type declarations[6],

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. We do not do anything with it yet, but still, you've just created a graph, in which:

- The out edges are stored in a std::vector

[3]In practice, these compiler error messages will be longer, bordering the unreadable

[4]if the function would throw because it cannot allocate this little piece of memory, you are in already in big trouble

[5][10], chapter 13.7, page 387

[6]Scott Meyers. C++ And Beyond 2012 session: 'Initial thoughts on Effective C++11'. 2012. 'Prefer auto to Explicit Type Declarations'

- 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. It stores its edges, out edges and vertices in a two different STL[7] containers. std::vector is the container you should use by default ([10] chapter 31.6, [11] chapter 76), as it has constant time look-up and back insertion. The std::list is used for storing the edges, as it is better suited at inserting elements at any position.

I use const to store the empty graph as we do not modify it. Correct use of const is called const-correct. Prefer to be const-correct ([9] chapter 7.9.3, [10] chapter 12.7, [7] item 3, [3] chapter 3, [11] item 15, [2] FAQ 14.05, [1] item 8, [4] 9.1.6).

I use auto...

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.

[7]Standard Template Library, the standard library

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, as displayed in figure 1. In a directed graph, an edge goes from a certain vertex, its source, to another (which may actually be the same), its target. A directed graph is shown in figure 2.

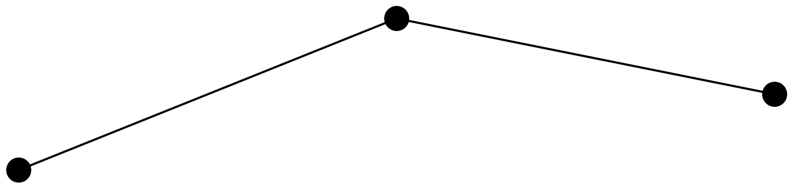

Figure 1: Example of an undirected graph

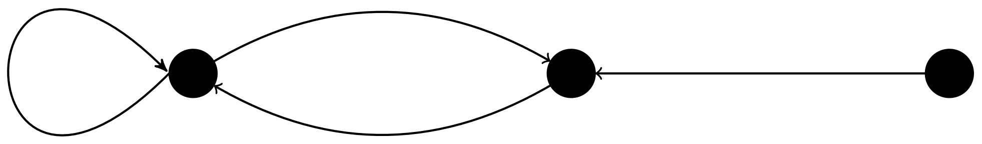

Figure 2: Example of a directed graph

2.3 Counting the number of vertices

Let's count all zero vertices of an empty graph!

Algorithm 5 Count the numbe of vertices

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

///Get the number of vertices a graph has
template <class graph>
int get_n_vertices(const graph& g) noexcept
{
  const int n{
    static_cast<int>(boost::num_vertices(g))
  };
  assert(n >= 0);
  return n;
}
```

The function 'get_n_vertices' takes the result of boost::num_vertices, converts it to int and checks if there was no range overflow. We do so, as one should prefer using int (over unsigned int) in an interface [4][8]. To do so, in the function body its first stament, the unsigned int[9] produced by boost::num_vertices get converted to an int using a static_cast. This static_cast cannot always be correct, as an unsigned int can have twice as high (but only positive) values.

[8]Chapter 9.2.2

[9]or '[some type]' to be precise

Luckily, this can be detected: if an unsigned int produces a negative int, it was too big to be stored as such. Using an unsigned int over a (signed) int for the sake of gaining that one more bit [9][10] should be avoided. The integer 'n' is initialized using list-initialization, which is preferred over the other initialization syntaxes ([10] chapter 17.7.6).

The assert statement checks if the conversion from unsigned int to int was successfull. If it was not, the program crashes. Use assert extensively ([9] chapter 24.5.18, [10] chapter 30.5, [11] chapter 68, [6] chapter 8.2, [5] hour 24, [4] chapter 2.6).

The function 'get_n_vertices' is demonstrated in algorithm 6, to measure the number of vertices of both the directed and undirected graph we are already able to create.

Algorithm 6 Demonstration of the 'get_n_vertices' function

```
#include <cassert>

#include "create_empty_directed_graph.h"
#include "create_empty_undirected_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_empty_undirected_graph();
  assert(get_n_vertices(h) == 0);
}
```

Note that the type of graph does not matter here. One can count the number of vertices of every graph, as all graphs have vertices. Boost.Graph is very good at detecting operations that are not allowed, during compile time.

2.4 Counting the number of edges

Let's count all zero edges of an empty graph!

This is very similar to the previous chapter, only it uses boost::num_edges instead.

[10]Chapter 4.4

Algorithm 7 Count the number of edges

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

///Get the number of edges a graph has
template <class graph>
int get_n_edges(const graph& g) noexcept
{
  const int n{
    static_cast<int>(boost::num_edges(g))
  };
  assert(n >= 0);
  return n;
}
```

For the rationale behind this, see the previous chapter.

The function 'get_n_edges' is demonstrated in algorithm 8, to measure the number of edges of an empty directed and undirected graph.

Algorithm 8 Demonstration of the 'get_n_edges' function

```
#include <cassert>

#include "create_empty_directed_graph.h"
#include "create_empty_undirected_graph.h"
#include "get_n_edges.h"

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

  const auto h = create_empty_undirected_graph();
  assert(get_n_edges(h) == 0);
}
```

2.5 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 9.

Algorithm 9 Adding a vertex to a graph

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

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

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 2.6, as we need these to add an edge. Algorithm 10 shows how to add a vertex to a directed and undirected graph.

Algorithm 10 Demonstration of the 'add_vertex' function

```
#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);
  assert(boost::num_vertices(g) == 1);

  auto h = create_empty_directed_graph();
  add_vertex(h);
  assert(boost::num_vertices(h) == 1);
}
```

This demonstration code creates two empty graphs, adds one vertex to each and then asserts that the number of vertices in each graph is one. This works for both types of graphs, as all graphs have vertices.

2.6 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 2.8 (vetrex iterators are obtained in chapter

Vertex descriptors are used to:

- add and edge between two vertices, see chapter 2.9
- obtain properties of vertex a vertex, for example the vertex its out degrees (chapter 27), the vertex its name (chapter 41), or a custom vertex property (chapter 89)

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

2.7 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 11. 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 11 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) noexcept
{
  return boost::vertices(g);
}
```

This is a somewhat trivial function, as it forwards the function call to boost::vertices.

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 12 demonstrates of the 'get_vertices' function, by showing that the vertex iterators of an empty graph point to the same location.

Algorithm 12 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);
}
```

2.8 Get all vertex descriptors

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

Vertex descriptors are obtained from dereferencing vertex iterators. Algorithm 13 shows how to obtain all vertex descriptors from a graph.

Algorithm 13 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;
  using vd = typename graph_traits<graph>::
      vertex_descriptor;

  std::vector<vd> vds;
  const auto vis = vertices(g); //No boost::vertices
  auto i = vis.first;
  const auto j = vis.second;
  for ( ; i!=j; ++i) {
    vds.emplace_back(*i);
  }
  return vds;
}
```

This is the first more complex piece of code. In the first lines, some 'using' statements allow for shorter type names. The function 'vertices' (not boost::vertices!) returns a vertex iterator pair. The two iterators are extracted, of which the first iterator, 'i', points to the first vertex, and the second, 'j', points to beyond the last vertex. In the for-loop, 'i' loops from begin to end. Dereferencing it produces a vertex descriptor, which is stored in the std::vector using emplace_back. Prefer using emplace_back ([10] chapter 31.6, items 25 and 27).

The 'get_vertex_descriptors' function shows an important concept of the Boost.Graph library: boost::verticesAlgorithm 14 demonstrates that an empty graph has no vertex descriptors.

Algorithm 14 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());
}
```

Because all graphs have (vertices and thus) vertex descriptors, the type of graph is unimportant for this code to compile.

2.9 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 2.6). Algorithm 15 adds two vertices to a graph, and connects these two using boost::add_edge:

Algorithm 15 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) noexcept
{
  const auto vd_a = boost::add_vertex(g);
  const auto vd_b = boost::add_vertex(g);
  const auto aer = boost::add_edge(
    vd_a, // Source/from
    vd_b, // Target/to
    g
  );

  assert(aer.second);
}
```

Algorithm 15 shows how to add an isolated edge to a graph (instead of allowing for graphs with higher connectivities). First, two vertices are created, using the function 'boost::add_vertex'. 'boost::add_vertex' returns a vertex descriptor (which I prefix with 'vd'), both of which are stored. The vertex descriptors are used to add an edge to the graph, using 'boost::add_edge'. 'boost::add_edge' returns returns a std::pair, consisting of an edge descriptor and a boolean success indicator. The success of adding the edge is checked by an assert statement. Here 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 16, in which an edge is added to both a directed and undirected graph.

Algorithm 16 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);
}
```

The graph type is unimportant: as all graph types have vertices and edges, edges can be added without possible compile problems.

2.10 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 2.12) and a boolean indicating insertion success.

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

2.11 Getting the edges

You cannot get the edges directly. Instead, 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 (its descriptor, to be precise). In this tutorial, edge iterator pairs have named prefixed with 'eip_', for example 'eip_1'. Algoritm 17 shows how to obtain these:

Algorithm 17 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) noexcept
{
  return boost::edges(g);
}
```

This is a somewhat trivial function, as all it does is forward to function call to 'edges' (not boost::edges!) 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 18 demonstrates 'get_edges' by showing that both iterators of the edge iterator pair point to the same location, when the graph is empty.

Algorithm 18 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);
}
```

2.12 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'.

2.13 Get all edge descriptors

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

Algorithm 19 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;
  using ed = typename graph_traits<graph>::
      edge_descriptor;

  std::vector<ed> eds;
  const auto ei = edges(g); //Not boost::edges
  auto i = ei.first;
  const auto j = ei.second;

  for ( ; i!=j; ++i) {
    eds.emplace_back(*i);
  }
  return eds;
}
```

This function is a repeat of getting all the vertex descriptors (see chapter 2.8), but using 'edges' (not boost::edges!) instead.

Algorithm 20 demonstrates the 'get_edge_descriptor', by showing that empty graphs do not have any edge descriptors.

Algorithm 20 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());
}
```

2.14 Creating K_2, a fully connected undirected 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 3.

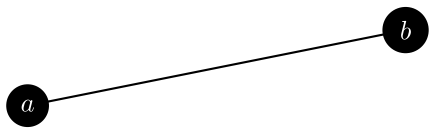

Figure 3: K_2: a fully connected graph with two vertices named a and b

To create K_2, the following code can be used:

Algorithm 21 Creating K_2 as depicted in figure 3

```
#include <boost/graph/adjacency_list.hpp>
#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 2.6) 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 2.10), 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 22 demonstrates how to 'create_k2_graph' and checks if it has the correct amount of edges and vertices.

Algorithm 22 Demonstration of 'create_k2_graph'

```
#include <cassert>

#include "create_k2_graph.h"

void create_k2_graph_demo() noexcept
{
  const auto g = create_k2_graph();
  assert(boost::num_vertices(g) == 2);
  assert(boost::num_edges(g) == 1);
}
```

Running a bit ahead, this graph can be converted to a .dot file (using algorithm 29) created is displayed in algorithm 23:

Algorithm 23 .dot file created from the create_k2_graph function (algorithm 21)

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

The .svg file of this graph is shown in figure 4:

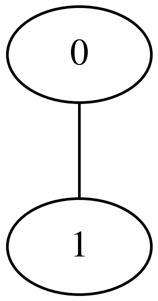

Figure 4: .svg file created from the 'create_k2_graph' function (algorithm 21) and converted to .svg using the 'convert_dot_to_svg' function (algorithm 124)

2.15 Creating a directed graph

Finally, we are going to create a directed graph!

This directed graph is a two-state Markov chain, with two vertices and four edges, as depicted in figure 5:

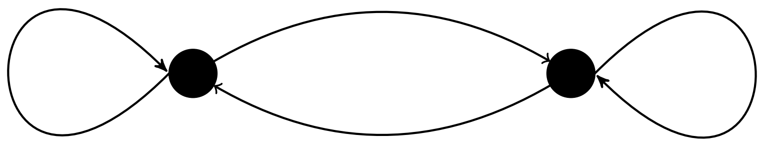

Figure 5: The two-state Markov chain

Note that directed graphs can have edges that start and end in the same vertex. These are called self-loops.

To create this two-state Markov chain, the following code can be used:

Algorithm 24 Creating K_2 as depicted in figure 5

```
#include <cassert>
#include <boost/graph/adjacency_list.hpp>
#include "create_empty_directed_graph.h"

boost::adjacency_list<>
create_markov_chain_graph() noexcept
{
  auto g = create_empty_directed_graph();
  const auto vd_a = boost::add_vertex(g);
  const auto vd_b = boost::add_vertex(g);
  const auto aer_aa = boost::add_edge(vd_a, vd_a, g);
  assert(aer_aa.second);
  const auto aer_ab = boost::add_edge(vd_a, vd_b, g);
  assert(aer_ab.second);
  const auto aer_ba = boost::add_edge(vd_b, vd_a, g);
  assert(aer_ba.second);
  const auto aer_bb = boost::add_edge(vd_b, vd_b, g);
  assert(aer_bb.second);
  return g;
}
```

To save defining the type, we call the 'create_empty_directed_graph' function. The vertex descriptors (see chapter 2.6) 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 2.10), 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 25 demonstrates the 'create_markov_chain_graph' function and checks if it has the correct amount of edges and vertices.

Algorithm 25 Demonstration of the 'create_markov_chain_graph'

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

#include "create_markov_chain_graph.h"

void create_markov_chain_graph_demo() noexcept
{
  const auto g = create_markov_chain_graph();
  assert(boost::num_vertices(g) == 2);
  assert(boost::num_edges(g) == 4);
}
```

Running a bit ahead, this graph can be converted to a .dot file (using algorithm 29) created is displayed in algorithm 26:

Algorithm 26 .dot file created from the 'create_markov_chain_graph' function (algorithm 24)

```
digraph G {
0;
1;
0->0 ;
0->1 ;
1->0 ;
1->1 ;
}
```

The .svg file of this graph is shown in figure 6:

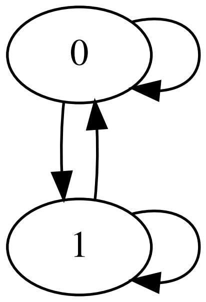

Figure 6: .svg file created from the 'create_markov_chain_graph' function (algorithm 24) and converted to .svg using the 'convert_dot_to_svg' function (algorithm 124)

2.16 Getting the vertices' out degree

As a bonus chapter, let's measure the out degree of all vertices in a graph. The out degree of a vertex is the number of edges that originate at it.

Algorithm 27 Get the vertices' out degrees

```
#include <vector>

template <typename graph>
std::vector<int> get_vertex_out_degrees(const graph& g)
    noexcept
{
  std::vector<int> v;
  const auto vis = vertices(g);
  auto i = vis.first;
  const auto j = vis.second;
  for ( ; i!=j; ++i) {
    v.emplace_back(
      out_degree(*i,g) //Not boost::out_degree
    );
  }
  return v;
}
```

The structure of this algorithm is similar to get_vertex_descriptors (algorithm 13), except that the out degrees from the vertex descriptors are stored. The out degree of a vertex iterator is obtained from the function 'out_degree' (not boost::out_degree!).

Albeit K_2 is a simple graph, we can use it to demonstrate 'get_vertex_out_degrees' on, as shown in algorithm 28.

Algorithm 28 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);
}
```

2.17 Storing a graph as a .dot

Graph are easily saved to a file, thanks to Graphviz. Graphviz (short for Graph Visualization Software) is a package of open-source tools for drawing graphs. It uses the DOT language for describing graphs, and these are commonly stored in (plain-text) .dot files:

Algorithm 29 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) noexcept
{
  std::ofstream f(filename);
  boost::write_graphviz(f,g);
}
```

All the code does is create an std::ofstream (an output-to-file stream) and use boost::write_graphviz to write the DOT description of our graph to that stream. Instead of 'std::ofstream', one could use std::cout (a related output stream) to display the DOT language on screen directly.

Algorithm 30 shows how to use the 'save_graph_to_dot' function:

Algorithm 30 Demonstration of the 'save_graph_to_dot' function

```
#include "create_k2_graph.h"
#include "create_named_vertices_k2_graph.h"
#include "save_graph_to_dot.h"

void save_graph_to_dot_demo() noexcept
{
  const auto g = create_k2_graph();
  save_graph_to_dot(g,"save_graph_to_dot_k2_graph.dot");

  const auto h = create_named_vertices_k2_graph();
  save_graph_to_dot(h,"
      save_graph_to_dot_named_vertices_k2_graph.dot");
}
```

When using the 'create_k2_graph' function (algorithm 21) to create a K_2 graph, the .dot file created is displayed in algorithm 31:

Algorithm 31 .dot file created from the create_k2_graph function (algorithm 21)

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

From the .dot file one can already see that the graph is undirected, because:

- The first word, 'graph', denotes an undirected graph (where 'digraph' would have indicated a directional graph)
- The edge between 0 and 1 is written as '–' (where directed connections would be written as '->', '<-' or '<>')

This .dot file corresponds to figure 7:

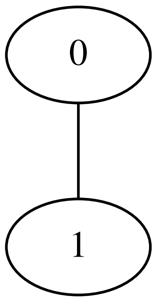

Figure 7: .svg file created from the 'create_k2_graph' function (algorithm 21) and converted to .svg using the 'convert_dot_to_svg' function (algorithm 124)

Also this figure shows that the graph in undirected, otherwise the edge would have one or two arrow heads.

If you used the create_named_vertices_k2_graph function (algorithm 43) to produce a K_2 graph with named vertices, you see that the .dot file does not have stored the vertex names:

Algorithm 32 .dot file created from the create_named_vertices_k2_graph function (algorithm 43)

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

So, the 'save_graph_to_dot' function (algorithm 29) saves only the structure of the graph.

2.18 Loading an undirected graph from a .dot

Before loading a graph from file, one needs to specify a type of graph. In this example, an undirected graph is loaded, as shown in algorithm 33:

Algorithm 33 Loading an undirected graph from a .dot file

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

#include "create_empty_undirected_graph.h"
#include "is_regular_file.h"

boost::adjacency_list<
  boost::vecS,
  boost::vecS,
  boost::undirectedS
>
load_undirected_graph_from_dot(
  const std::string& dot_filename
)
{
  assert(is_regular_file(dot_filename));
  std::ifstream f(dot_filename.c_str());
  auto g = create_empty_undirected_graph();
  boost::dynamic_properties p(boost::
      ignore_other_properties);
  boost::read_graphviz(f,g,p);
  return g;
}
```

In this algorithm, first it is checked if the file to load exists. Then an empty undirected graph is created. Next to this, a boost::dynamic_properties is created with the 'boost::ignore_other_properties' in its constructor (using a default constructor here results in the run-time error 'property not found: node_id', see chapter 10.5). From this and the empty graph, 'boost::read_graphviz' is called to build up the graph.

Algorithm 34 shows how to use the 'load_undirected_graph_from_dot' function:

Algorithm 34 Demonstration of the 'load_undirected_graph_from_dot' function

```
#include "create_k2_graph.h"
#include "load_undirected_graph_from_dot.h"
#include "save_graph_to_dot.h"

void load_undirected_graph_from_dot_demo() noexcept
{
  const auto g = create_k2_graph();
  const std::string filename{"
     load_undirected_graph_from_dot.dot"};
  save_graph_to_dot(g, filename);
  const auto h = load_undirected_graph_from_dot(filename)
     ;
  assert(boost::num_edges(g) == boost::num_edges(h));
  assert(boost::num_vertices(g) == boost::num_vertices(h)
     );
}
```

This demonstration shows how the K_2 graph is created using the 'create_k2_graph' function (algorithm 21), saved and then loaded. The loaded graph should be a K_2 graph.

Figure 8 shows that the graph loaded can be converted to the correct graph (which should be identical to figure 7):

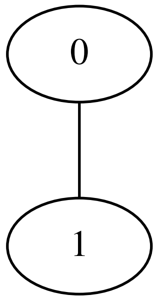

Figure 8: .svg file created from the 'load_undirected_graph_from_dot_demo' function (algorithm 34) and converted to .svg using the 'convert_dot_to_svg' function (algorithm 124)

2.19 Loading an directed graph from a .dot

When loading a graph from file, one needs to specify a type of graph. In this example, an directed graph is loaded, as shown in algorithm 35:

Algorithm 35 Loading a directed graph from a .dot file

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

#include "create_empty_directed_graph.h"
#include "is_regular_file.h"

boost::adjacency_list<
  boost::vecS,
  boost::vecS,
  boost::directedS
>
load_directed_graph_from_dot(
  const std::string& dot_filename
)
{
  assert(is_regular_file(dot_filename));
  std::ifstream f(dot_filename.c_str());
  auto g = create_empty_directed_graph();
  boost::dynamic_properties p(boost::
      ignore_other_properties);
  boost::read_graphviz(f,g,p);
  return g;
}
```

In this algorithm, first it is checked if the file to load exists. Then an empty directed graph is created. Next to this, a boost::dynamic_properties is created with the 'boost::ignore_other_properties' in its constructor (using a default constructor here results in the run-time error 'property not found: node_id', see chapter 10.5). From this and the empty graph, 'boost::read_graphviz' is called to build up the graph.

Algorithm 36 shows how to use the 'load_directed_graph_from_dot' function:

Algorithm 36 Demonstration of the 'load_directed_graph_from_dot' function

```
#include "create_markov_chain_graph.h"
#include "load_directed_graph_from_dot.h"
#include "save_graph_to_dot.h"

void load_directed_graph_from_dot_demo() noexcept
{
  const auto g = create_markov_chain_graph();
  const std::string filename{"
     load_directed_graph_from_dot.dot"};
  save_graph_to_dot(g, filename);
  const auto h = load_directed_graph_from_dot(filename);
  assert(boost::num_edges(g) == boost::num_edges(h));
  assert(boost::num_vertices(g) == boost::num_vertices(h)
     );
}
```

This demonstration shows how the Markov chain is created using the 'create_markov_chain_graph' function (algorithm 24), saved and then loaded. The loaded graph should be a directed graph similar to the Markov chain.

Figure 9 shows that the graph loaded can be converted to the correct graph (which should be identical to figure 7):

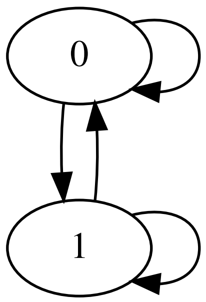

Figure 9: .svg file created from the 'load_directed_graph_from_dot_demo' function (algorithm 36) and converted to .svg using the 'convert_dot_to_svg' function (algorithm 124)

3 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 name. This name will be of the std::string data type, but other types are possible as well. There are many more built-in properties edges and nodes can have (see the boost/graph/properties.hpp file for these).

In this chapter, we will build the following graphs:

- An empty (undirected) graph that allows for vertices with names: see chapter 3.1
- K_2 with named vertices: see chapter 3.4
- An empty (undirected) graph that allows for edges and vertices with names: see chapter 3.5
- $K_3$with named edges and vertices: see chapter 3.8

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

- Adding a named vertex: see chapter 3.2
- Getting the vertices' names: see chapter 3.3
- Adding an named edge: see chapter 3.6
- Getting the edges' names: see chapter 3.7

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

3.1 Creating an empty undirected graph with named vertices

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

Algorithm 37 Creating an empty undirected 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_undirected_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 38 shows how to create this graph. Note that all the earlier functions defined in this tutorial keep working as expected.

Algorithm 38 Demonstration if the 'create_empty_undirected_named_vertices_graph' function

```
#include <cassert>

#include "create_empty_undirected_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_undirected_named_vertices_graph_demo()
    noexcept
{
  const auto g =
      create_empty_undirected_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());
}
```

3.2 Add a vertex with a name

Adding a vertex without a name was trivially easy (see chapter 9). Adding a vertex with a name is less easy:

Algorithm 39 Add a vertex with a name

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

template <typename graph>
void add_named_vertex(const std::string& name, graph& g)
    noexcept
{
  const auto vd_a = boost::add_vertex(g);
  auto vertex_name_map = 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[11] , multiple things need to be done. When adding a new vertex to the graph, the vertex descriptor (as describes in chapter 2.6) 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 40.

Algorithm 40 Demonstration of 'add_named_vertex'

```
#include <cassert>

#include "add_named_vertex.h"
#include "create_empty_undirected_named_vertices_graph.h"
#include "get_vertex_descriptors.h"

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

3.3 Getting the vertices' names

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

[11]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 41 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)
    noexcept
{
  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 10.1).

Algorithm 42 shows how to add two named vertices and how to get their names.

Algorithm 42 Demonstration of 'get_vertex_names'

```
#include <cassert>

#include "add_named_vertex.h"
#include "create_empty_undirected_named_vertices_graph.h"
#include "get_vertex_names.h"

void get_vertex_names_demo() noexcept
{
  auto g = create_empty_undirected_named_vertices_graph()
      ;
  const std::string vertex_name_1{"Chip"};
  const std::string vertex_name_2{"Chap"};
  add_named_vertex(vertex_name_1, g);
  add_named_vertex(vertex_name_2, g);
  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);
}
```

3.4 Creating K_2 with named vertices

We extend K_2 of chapter 2.14 by naming the vertices 'from' and 'to', as depicted in figure 10:

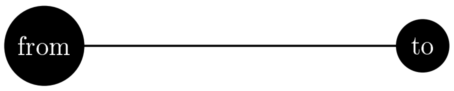

Figure 10: K_2: a fully connected graph with two vertices with the text 'from' and 'to'

To create K_2, the following code can be used:

Algorithm 43 Creating K_2 as depicted in figure 10

```
#include "create_named_vertices_k2_graph.h"

#include "create_named_vertices_k2_graph.impl"

#include "create_named_vertices_k2_graph_demo.impl"

#include <cassert>
#include <iostream>
#include "get_n_edges.h"
#include "get_n_vertices.h"
#include "get_edges.h"
#include "get_vertices.h"
#include "get_edge_descriptors.h"
#include "get_vertex_descriptors.h"
#include "get_vertex_names.h"
#include "create_named_vertices_k2_graph.h"

void create_named_vertices_k2_graph_test() noexcept
{
  const auto g = create_named_vertices_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);

  assert(get_n_edges(g) == 1);
  assert(get_n_vertices(g) == 2);
  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);

  create_named_vertices_k2_graph_demo();
  std::cout << __func__ << ": OK" << '\n';
}
```

Most of the code is a repeat of algorithm 21. 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 44 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);
}
```

3.5 Creating an empty undirected graph with named edges and vertices

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

Algorithm 45 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_undirected_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 46 shows how to create this graph. Note that all the earlier functions defined in this tutorial keep working as expected.

Algorithm 46 Demonstration if the 'create_empty_undirected_named_edges_and_vertices_graph' function

```
#include <cassert>

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

void
    create_empty_undirected_named_edges_and_vertices_graph_demo
    () noexcept
{
  auto g =
      create_empty_undirected_named_edges_and_vertices_graph
      ();
  add_named_edge("Reed", g);
  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);
}
```

3.6 Adding a named edge

Adding an edge with a name:

Algorithm 47 Add a vertex with a name

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

#include <cassert>

template <typename graph>
void add_named_edge(const std::string& edge_name, graph&
    g) noexcept
{
  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 = get(boost::edge_name,g);
  edge_name_map[aer.first] = edge_name;
}
```

In this code snippet, the edge descriptor (see chapter 2.12 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 48 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 10.1).

Algorithm 48 Demonstration of the 'add_named_edge' function

```
#include <cassert>

#include "add_named_edge.h"
#include "
    create_empty_undirected_named_edges_and_vertices_graph
    .h"
#include "get_n_edges.h"

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

3.7 Getting the edges' names

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

Algorithm 49 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)
    noexcept
{
  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 50 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 10.1).

Algorithm 50 Demonstration of the 'get_edge_names' function

```
#include <cassert>

#include "add_named_edge.h"
#include "
    create_empty_undirected_named_edges_and_vertices_graph
    .h"
#include "get_edge_names.h"

void get_edge_names_demo() noexcept
{
  auto g =
      create_empty_undirected_named_edges_and_vertices_graph
      ();
  const std::string edge_name_1{"Eugene"};
  const std::string edge_name_2{"Another␣Eugene"};
  add_named_edge(edge_name_1, g);
  add_named_edge(edge_name_2, g);
  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);
}
```

3.8 Creating K_3 with named edges and vertices

We extend the graph K_2 with named vertices of chapter 3.4 by adding names to the edges, as depicted in figure 11:

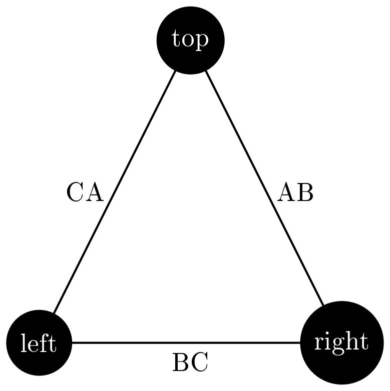

Figure 11: K_3: a fully connected graph with three named edges and vertices

To create K_3, the following code can be used:

Algorithm 51 Creating K_3 as depicted in figure 11

```
#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_undirected_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 = 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 = 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 43. In the end, the edge names are obtained as a boost::property_map and set. Algorithm 52 shows how to create the graph and measure its edge and vertex names.

Algorithm 52 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);
}
```

4 Working with graphs with named edges and vertices

Measuring simple traits of the graphs created allows

- Check if there exists a vertex with a certain name: chapter 4.1
- Find a (named) vertex by its name: chapter 4.2
- Get a (named) vertex its degree, in degree and out degree: chapter: 4.3
- Get a (named) vertex its name from its vertex descriptor: chapter 4.4
- Set a (named) vertex its name using its vertex descriptor: chapter 4.5
- Setting all vertices' names: chapter 4.6
- Clear a named vertex its edges: chapter 4.7

- Remove a named vertex: chapter 4.8
- Check if there exists an edge with a certain name: chapter 4.9
- Find a (named) edge by its name: chapter 4.10
- Get a (named) edge its name from its edge descriptor: chapter
- Set a (named) edge its name using its edge descriptor: chapter
- Remove a named edge: chapter
- Storing an undirected graph with named vertices as a .dot: chapter 4.14
- Loading an undirected graph with named vertices as a .dot: chapter
- Storing an directed graph with named vertices as a .dot: chapter
- Storing a graph with named edges and vertices as a .dot: chapter 4.15

Especially the first paragraph is important: 'find_first_vertex_by_name' shows how to obtain a vertex descriptor, which is used in later algorithms.

4.1 Check if there exists a vertex with a certain name

Before modifying our vertices, let's first determine if we can find a vertex by its name in a graph. After obtaing a name map, we obtain the vertex iterators, dereference these to obtain the vertex descriptors and then compare each vertex its name with the one desired.

Algorithm 53 Find if there is vertex with a certain name

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

template <typename graph>
bool has_vertex_with_name(
  const std::string& name,
  const graph& g
) noexcept
{
  const auto vertex_name_map = get(boost::vertex_name, g)
      ;

  for (auto p = vertices(g);
    p.first != p.second;
    ++p.first) {
    if (get(vertex_name_map, *p.first) == name) {
      return true;
    }
  }
  return false;
}
```

This function can be demonstrated as in algorithm 54, where a certain name cannot be found in an empty graph. After adding the desired name, it is found.

Algorithm 54 Demonstration of the 'has_vertex_with_name' function

```
#include <cassert>

#include "add_named_vertex.h"
#include "create_empty_undirected_named_vertices_graph.h"
#include "has_vertex_with_name.h"

void has_vertex_with_name_demo() noexcept
{
  auto g = create_empty_undirected_named_vertices_graph()
      ;
  assert(!has_vertex_with_name("Felix",g));
  add_named_vertex("Felix",g);
  assert(has_vertex_with_name("Felix",g));
}
```

Note that this function only finds if there is at least one vertex with that name: it does not tell how many vertices with that name exist in the graph.

4.2 Find a vertex by its name

Where STL functions work with iterators, here we obtain a vertex descriptor (see chapter 2.6) to obtain a handle to the desired vertex. Algorithm 55 shows how to obtain a vertex descriptor to the first (name) vertex found with a specific name.

Algorithm 55 Find the first vertex by its name

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

template <typename graph>
typename boost::graph_traits<graph>::vertex_descriptor
find_first_vertex_with_name(
  const std::string& name,
  const graph& g
) noexcept
{
  const auto vertex_name_map = get(boost::vertex_name,g);

  for (auto p = vertices(g);
    p.first != p.second;
    ++p.first) {
    const std::string s{
      get(vertex_name_map, *p.first)
    };
    if (s == name) { return *p.first; }
  }
  return *vertices(g).second;

}
```

With the vertex descriptor obtained, one can read and modify the vertex and the edges surrounding it. Algorithm 56 shows some examples of how to do so.

Algorithm 56 Demonstration of the 'find_first_vertex_by_name' function

```
#include <cassert>

#include "create_named_vertices_k2_graph.h"
#include "find_first_vertex_with_name.h"

void find_first_vertex_with_name_demo() noexcept
{
  const auto g = create_named_vertices_k2_graph();
  const auto vd = find_first_vertex_with_name("from", g);
  assert(boost::out_degree(vd,g) == 1);
  assert(boost::in_degree(vd,g) == 1);
}
```

4.3 Get a (named) vertex its degree, in degree and out degree

We already obtained all out degrees of all vertices in chapter 2.16 by just collecting all vertex descriptors. Here, we will search for a vertex with a certain name, obtain its vertex descriptor and find the number of connections it has. The number of connections is called the 'degree' of the vertex. There are three types of degrees:

- in degree: the number of incoming connections, using boost::in_degree
- out degree: the number of outgoing connections, using boost::in_degree
- degree: sum of the in degree and out degree, using boost::in_degree

With a vertex descriptor, we can read a vertex its types of degrees. Algorithm 27 shows how to find a vertex, obtain its vertex descriptor and then obtain the out degree from it.

Algorithm 57 Get the first vertex with a certain name its out degree from its vertex descriptor

```
#include <cassert>
#include <string>
#include <boost/graph/adjacency_list.hpp>
#include "find_first_vertex_with_name.h"
#include "has_vertex_with_name.h"

template <typename graph>
int get_first_vertex_with_name_out_degree(
  const std::string& name,
  const graph& g) noexcept
{
  assert(has_vertex_with_name(name, g));
  const auto vd = find_first_vertex_with_name(name, g);
  return static_cast<int>(boost::out_degree(vd, g));
}
```

Algorithm 42 shows how to use this function.

Algorithm 58 Demonstration of the 'get_first_vertex_with_name_out_degree' function

```
#include <cassert>
#include "create_named_vertices_k2_graph.h"
#include "get_first_vertex_with_name_out_degree.h"

void get_first_vertex_with_name_out_degree_demo()
    noexcept
{
  const auto g = create_named_vertices_k2_graph();
  assert(get_first_vertex_with_name_out_degree("from", g)
       == 1);
  assert(get_first_vertex_with_name_out_degree("to", g)
      == 1);
}
```

4.4 Get a (named) vertex its name from its vertex descriptor

This may seem a trivial paragraph, as chapter 3.3 describes the 'get_vertex_names' algorithm, in which we get all vertices' names. But it does not allow to first find a vertex of interest and subsequently getting only that one its name.

To obtain the name from a vertex descriptor, one needs to pull out the name map and then look up the vertex of interest (I like to compare it as such: the vertex descriptor is a last name, the name map is a phone book, the desired info a phone number).

Algorithm 59 Get a vertex its name from its vertex descriptor

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

template <typename graph>
std::string get_vertex_name(
  const typename boost::graph_traits<graph>::
     vertex_descriptor& vd,
  const graph& g
) noexcept
{
  const auto vertex_name_map = get(boost::vertex_name,g);
  return vertex_name_map[vd];
}
```

To use 'get_vertex_name', one first needs to obtain a vertex descriptor. Algorithm 42 shows a simple example.

Algorithm 60 Demonstration if the 'get_vertex_name' function

```
#include <cassert>

#include "add_named_vertex.h"
#include "create_empty_undirected_named_vertices_graph.h"
#include "find_first_vertex_with_name.h"
#include "get_vertex_name.h"

void get_vertex_name_demo() noexcept
{
  auto g = create_empty_undirected_named_vertices_graph()
     ;
  const std::string name{"Dex"};
  add_named_vertex(name, g);
  const auto vd = find_first_vertex_with_name(name,g);
  assert(get_vertex_name(vd,g) == name);
}
```

4.5 Set a (named) vertex its name from its vertex descriptor

If you know how to get the name from a vertex descriptor, setting it is just as easy, as shown in algorithm 61.

Algorithm 61 Set a vertex its name from its vertex descriptor

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

template <typename graph>
void set_vertex_name(
  const std::string& name,
  const typename boost::graph_traits<graph>::
      vertex_descriptor& vd,
  graph& g
) noexcept
{
  auto vertex_name_map = get(boost::vertex_name,g);
  vertex_name_map[vd] = name;
}
```

To use 'set_vertex_name', one first needs to obtain a vertex descriptor. Algorithm 62 shows a simple example.

Algorithm 62 Demonstration if the 'set_vertex_name' function

```
#include <cassert>

#include "add_named_vertex.h"
#include "create_empty_undirected_named_vertices_graph.h"
#include "find_first_vertex_with_name.h"
#include "get_vertex_name.h"
#include "set_vertex_name.h"

void set_vertex_name_demo() noexcept
{
  auto g = create_empty_undirected_named_vertices_graph()
      ;
  const std::string old_name{"Dex"};
  add_named_vertex(old_name, g);
  const auto vd = find_first_vertex_with_name(old_name,g)
      ;
  assert(get_vertex_name(vd,g) == old_name);
  const std::string new_name{"Diggy"};
  set_vertex_name(new_name, vd, g);
  assert(get_vertex_name(vd,g) == new_name);
}
```

4.6 Setting all vertices' names

When the vertices of a graph have named vertices and you want to set all their names at once:

Algorithm 63 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
) noexcept
{
  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.

4.7 Clear the edges of a named vertex

A vertex descriptor can be used to clear all in/out/both edges connected to a vertex. It is necessary to remove these connections before the vertex itself can be removed. There are three functions to remove the edges connected to a vertex:

- boost::clear_vertex: removes all edges to and from the vertex
- boost::clear_out_edges: removes all outgoing edges from the vertex (in directed graphs only, else you will get a 'error: no matching function for call to clear_out_edges', as described in chapter 10.2)
- boost::clear_in_edges: removes all incoming edges from the vertex (in

directed graphs only, else you will get a 'error: no matching function for call to clear_in_edges', as described in chapter 10.3)

In the algorithm 'clear_first_vertex_with_name' the 'boost::clear_vertex' algorithm is used, as the graph used is undirectional:

Algorithm 64 Clear the first vertex with a certain name

```
#include <string>
#include <boost/graph/adjacency_list.hpp>
#include "find_first_vertex_with_name.h"
#include "has_vertex_with_name.h"

template <class graph>
void clear_first_vertex_with_name(
  const std::string& name,
  graph& g
) noexcept
{
  assert(has_vertex_with_name(name,g));
  const auto vd = find_first_vertex_with_name(name,g);
  boost::clear_vertex(vd,g);
}
```

Algorithm 65 shows the clearing of the first named vertex found.

Algorithm 65 Demonstration of the 'clear_first_vertex_with_name' function

```
#include <cassert>

#include "clear_first_vertex_with_name.h"
#include "create_named_vertices_k2_graph.h"
#include "get_n_edges.h"

void clear_first_vertex_with_name_demo() noexcept
{
  auto g = create_named_vertices_k2_graph();
  assert(get_n_edges(g) == 1);
  clear_first_vertex_with_name("from",g);
  assert(get_n_edges(g) == 0);
}
```

4.8 Remove a named vertex

A vertex descriptor can be used to remove a vertex from a graph. It is necessary to remove these connections (e.g. using clear_first_vertex_with_name', algorithm 64) before the vertex itself can be removed.

Removing a named vertex goes as follows: use the name of the vertex to get a first vertex descriptor, then call 'boost::remove_vertex', shown in algorithm 64.

Algorithm 66 Remove the first vertex with a certain name

```
#include <string>
#include <boost/graph/adjacency_list.hpp>
#include "find_first_vertex_with_name.h"
#include "has_vertex_with_name.h"

template <class graph>
void remove_first_vertex_with_name(
  const std::string& name,
  graph& g
) noexcept
{
  assert(has_vertex_with_name(name,g));
  const auto vd = find_first_vertex_with_name(name,g);
  assert(boost::degree(vd,g) == 0);
  boost::remove_vertex(vd,g);
}
```

Algorithm 67 shows the removal of the first named vertex found.

Algorithm 67 Demonstration of the 'remove_first_vertex_with_name' function

```
#include <cassert>

#include "clear_first_vertex_with_name.h"
#include "create_named_vertices_k2_graph.h"
#include "get_n_edges.h"
#include "get_n_vertices.h"
#include "remove_first_vertex_with_name.h"

void remove_first_vertex_with_name_demo() noexcept
{
  auto g = create_named_vertices_k2_graph();
  clear_first_vertex_with_name("from",g);
  remove_first_vertex_with_name("from",g);
  assert(get_n_edges(g) == 0);
  assert(get_n_vertices(g) == 1);
}
```

Again, be sure that the vertex removed does not have any connections!

4.9 Check if there exists an edge with a certain name

Before modifying our edges, let's first determine if we can find an edge by its name in a graph. After obtaing a name map, we obtain the edge iterators, dereference these to obtain the edge descriptors and then compare each edge its name with the one desired.

Algorithm 68 Find if there is an edge with a certain name

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

template <typename graph>
bool has_edge_with_name(
  const std::string& name,
  const graph& g
) noexcept
{
  const auto edge_name_map = get(boost::edge_name,g);

  for (auto p = edges(g);
    p.first != p.second;
    ++p.first) {
    if (get(edge_name_map, *p.first) == name) {
      return true;
    }
  }
  return false;
}
```

This function can be demonstrated as in algorithm 69, where a certain name cannot be found in an empty graph. After adding the desired name, it is found.

Algorithm 69 Demonstration of the 'has_edge_with_name' function

```
#include <cassert>

#include "add_named_edge.h"
#include "
    create_empty_undirected_named_edges_and_vertices_graph
    .h"
#include "has_edge_with_name.h"

void has_edge_with_name_demo() noexcept
{
  auto g =
      create_empty_undirected_named_edges_and_vertices_graph
      ();
  assert(!has_edge_with_name("Edward",g));
  add_named_edge("Edward",g);
  assert(has_edge_with_name("Edward",g));
}
```

Note that this function only finds if there is at least one edge with that name: it does not tell how many edges with that name exist in the graph.

4.10 Find an edge by its name

Where STL functions work with iterators, here we obtain an edge descriptor (see chapter 2.12) to obtain a handle to the desired edge. Algorithm 70 shows how to obtain an edge descriptor to the first (name) edge found with a specific name.

Algorithm 70 Find the first edge by its name

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

template <typename graph>
typename boost::graph_traits<graph>::edge_descriptor
find_first_edge_with_name(
  const std::string& name,
  const graph& g
) noexcept
{
  const auto edge_name_map = get(boost::edge_name,g);

  for (auto p = edges(g);
    p.first != p.second;
    ++p.first) {
    const std::string s{
      get(edge_name_map, *p.first)
    };
    if (s == name) { return *p.first; }
  }
  return *edges(g).second;
}
```

With the edge descriptor obtained, one can read and modify the graph. Algorithm 71 shows some examples of how to do so.

Algorithm 71 Demonstration of the 'find_first_edge_by_name' function

```
#include <cassert>

#include "create_named_edges_and_vertices_k3_graph.h"
#include "find_first_edge_with_name.h"

void find_first_edge_with_name_demo() noexcept
{
  const auto g = create_named_edges_and_vertices_k3_graph
      ();
  const auto ed = find_first_edge_with_name("AB", g);
  assert(boost::source(ed,g) != boost::target(ed,g));
}
```

4.11 Get a (named) edge its name from its edge descriptor

This may seem a trivial paragraph, as chapter 3.7 describes the 'get_edge_names' algorithm, in which we get all edges' names. But it does not allow to first find an edge of interest and subsequently getting only that one its name.

To obtain the name from an edgedescriptor, one needs to pull out the name map and then look up the edge of interest.

Algorithm 72 Get an edge its name from its edge descriptor

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

template <typename graph>
std::string get_edge_name(
  const typename boost::graph_traits<graph>::
      edge_descriptor& vd,
  const graph& g
) noexcept
{
  const auto edge_name_map = get(boost::edge_name,g);
  return edge_name_map[vd];
}
```

To use 'get_edge_name', one first needs to obtain an edge descriptor. Algorithm 42 shows a simple example.

Algorithm 73 Demonstration if the 'get_edge_name' function

```
#include <cassert>

#include "add_named_edge.h"
#include "
    create_empty_undirected_named_edges_and_vertices_graph
    .h"
#include "find_first_edge_with_name.h"
#include "get_edge_name.h"

void get_edge_name_demo() noexcept
{
  auto g =
      create_empty_undirected_named_edges_and_vertices_graph
      ();
  const std::string name{"Dex"};
  add_named_edge(name, g);
  const auto ed = find_first_edge_with_name(name,g);
  assert(get_edge_name(ed,g) == name);
}
```

4.12 Set a (named) edge its name from its edge descriptor

If you know how to get the name from an edge descriptor, setting it is just as easy, as shown in algorithm 74.

Algorithm 74 Set an edge its name from its edge descriptor

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

template <typename graph>
void set_edge_name(
  const std::string& name,
  const typename boost::graph_traits<graph>::
      edge_descriptor& vd,
  graph& g
) noexcept
{
  auto edge_name_map = get(boost::edge_name,g);
  edge_name_map[vd] = name;
}
```

To use 'set_edge_name', one first needs to obtain an edge descriptor. Algorithm 75 shows a simple example.

Algorithm 75 Demonstration if the 'set_edge_name' function

```
#include <cassert>

#include "add_named_edge.h"
#include "
    create_empty_undirected_named_edges_and_vertices_graph
    .h"
#include "find_first_edge_with_name.h"
#include "get_edge_name.h"
#include "set_edge_name.h"

void set_edge_name_demo() noexcept
{
  auto g =
      create_empty_undirected_named_edges_and_vertices_graph
      ();
  const std::string old_name{"Dex"};
  add_named_edge(old_name, g);
  const auto vd = find_first_edge_with_name(old_name,g);
  assert(get_edge_name(vd,g) == old_name);
  const std::string new_name{"Diggy"};
  set_edge_name(new_name, vd, g);
  assert(get_edge_name(vd,g) == new_name);
}
```

4.13 Removing a named edge

There are two ways to remove an edge:

1. Get an edge descriptor and call 'boost::remove_edge' on that descriptor: chapter 4.13.1
2. Get two vertex descriptors and call 'boost::remove_edge' on those two descriptors: chapter 4.13.2

4.13.1 Removing the first edge with a certain name

An edge descriptor can be used to remove an edge from a graph.

Removing a named edge goes as follows: use the name of the edge to get a first edge descriptor, then call 'boost::remove_edge', shown in algorithm 64.

Algorithm 76 Remove the first edge with a certain name

```
#include <string>
#include <boost/graph/adjacency_list.hpp>
#include "find_first_edge_with_name.h"
#include "has_edge_with_name.h"

template <class graph>
void remove_first_edge_with_name(
  const std::string& name,
  graph& g
) noexcept
{
  assert(has_edge_with_name(name,g));
  const auto vd = find_first_edge_with_name(name,g);
  boost::remove_edge(vd,g);
}
```

Algorithm 77 shows the removal of the first named edge found.

Algorithm 77 Demonstration of the 'remove_first_edge_with_name' function

```
#include <cassert>

#include "create_named_edges_and_vertices_k3_graph.h"
#include "get_n_edges.h"
#include "get_n_vertices.h"
#include "remove_first_edge_with_name.h"

void remove_first_edge_with_name_demo() noexcept
{
  auto g = create_named_edges_and_vertices_k3_graph();
  assert(get_n_edges(g) == 3);
  assert(get_n_vertices(g) == 3);
  remove_first_edge_with_name("AB",g);
  assert(get_n_edges(g) == 2);
  assert(get_n_vertices(g) == 3);
}
```

4.13.2 Removing the edge between two named vertices

Instead of looking for an edge descriptor, one can also remove an edge from two vertex descriptors (which is: the edge between the two vertices). Removing an edge between two named vertices named edge goes as follows: use the names of the vertices to get both vertex descriptors, then call 'boost::remove_edge' on those two, as shown in algorithm 64.

Algorithm 78 Remove the first edge with a certain name

```
#include <string>
#include <boost/graph/adjacency_list.hpp>
#include "find_first_vertex_with_name.h"
#include "has_vertex_with_name.h"
#include "has_edge_between_vertices.h"

template <typename graph>
void remove_edge_between_vertices_with_names(
  const std::string& name_1,
  const std::string& name_2,
  graph& g
) noexcept
{
  assert(has_vertex_with_name(name_1, g));
  assert(has_vertex_with_name(name_2, g));
  const auto vd_1 = find_first_vertex_with_name(name_1, g
     );
  const auto vd_2 = find_first_vertex_with_name(name_2, g
     );
  assert(has_edge_between_vertices(vd_1, vd_2, g));
  boost::remove_edge(vd_1, vd_2, g);
}
```

Algorithm 79 shows the removal of the first named edge found.

Algorithm 79 Demonstration of the 'remove_edge_between_vertices_with_names' function

```
#include <cassert>

#include "create_named_edges_and_vertices_k3_graph.h"
#include "get_n_edges.h"

void remove_edge_between_vertices_with_names_demo()
    noexcept
{
  auto g = create_named_edges_and_vertices_k3_graph();
  assert(get_n_edges(g) == 3);
  remove_edge_between_vertices_with_names("top","right",g
      );
  assert(get_n_edges(g) == 2);
}
```

4.14 Storing an directed/undirected graph with named vertices as a .dot

If you used the create_named_vertices_k2_graph function (algorithm 43) to produce a K_2 graph with named vertices, you can store these names additionally with algorithm 80:

Algorithm 80 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) noexcept
{
  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 81:

Algorithm 81 .dot file created from the create_named_vertices_k2_graph function (algorithm 43)

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

This .dot file corresponds to figure 12:

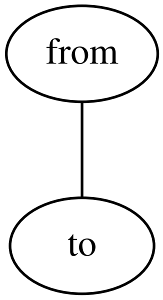

Figure 12: .svg file created from the create_k2_graph function (algorithm 43) and converted to .svg using the 'convert_dot_to_svg' function (algorithm 124)

If you used the create_named_edges_and_vertices_k3_graph function (algorithm 51) 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 82 .dot file created from the create_named_edges_and_vertices_k3_graph function (algorithm 51)

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

4.15 Storing an undirected graph with named vertices and edges as a .dot

If you used the create_named_edges_and_vertices_k3_graph function (algorithm 51) to produce a K_3 graph with named edges and vertices, you can store these names additionally with algorithm 83:

Algorithm 83 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 84:

Algorithm 84 .dot file created from the create_named_edges_and_vertices_k3_graph function (algorithm 43)

```
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 13:

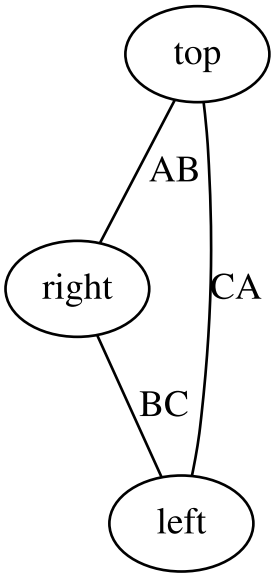

Figure 13: .svg file created from the create_named_edges_and_vertices_k3_graph function (algorithm 43) and converted to .svg using the 'convert_dot_to_svg' function (algorithm 124)

If you created a graph with edges more complex than just a name, you will still just write these to the .dot file. Chapter 6.10 shows how to write custom vertices to a .dot file.

So, the 'save_named_edges_and_vertices_graph_to_dot' function (algorithm 29) saves only the structure of the graph and its edge and vertex names.

5 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[12].

- An empty (undirected) graph that allows for custom vertices: see chapter 5.1
- $K_2$with custom vertices: see chapter 5.4
- An empty (undirected) graph that allows for custom edges and vertices: see chapter 5.5
- $K_3$with custom edges and vertices: see chapter 5.7

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

- Adding a custom vertex: see chapter 5.2
- Adding a custom edge: see chapter 5.6

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

5.1 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

5.1.1 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.

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

Algorithm 85 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.

5.1.2 Installing the new property

Installing a new property would have been easier, if 'more C++ compilers were standards conformant' ([8], chapter 3.6, footnote at page 52). Boost.Graph uses the BOOST_INSTALL_PROPERTY macro to allow using a custom property:

Algorithm 86 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.

5.1.3 Create the empty graph with custom vertices

Algorithm 87 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".

5.2 Add a custom vertex

Adding a custom vertex is very similar to adding a named vertex (chapter 3.2).

Algorithm 88 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(const my_vertex& v, graph& g)
    noexcept
{
  const auto vd_a = boost::add_vertex(g);
  const auto my_vertex_map = 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)').

5.3 Getting the vertices' my_vertexes[13]

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

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

Algorithm 89 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) noexcept
{
  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 10.1).

5.4 Creating K_2 with custom vertices

We reproduce the K_2 with named vertices of chapter 3.4 , but with our custom vertices intead:

Algorithm 90 Creating K_2 as depicted in figure 10

```
#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_vertexes_map = get(boost::vertex_custom_type,g)
      ;
  my_vertexes_map[vd_a]
    = my_vertex("from","source",0.0,0.0);
  my_vertexes_map[vd_b]
    = my_vertex("to","target",3.14,3.14);

  return g;
}
```

Most of the code is a slight modification of algorithm 43. In the end, the my_vertices are obtained as a boost::property_map and set with two custom my_vertex objects.

5.5 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

5.5.1 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 91 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.

5.5.2 Installing the new property

Installing a new property would have been easier, if 'more C++ compilers were standards conformant' ([8], chapter 3.6, footnote at page 52). Boost.Graph uses the BOOST_INSTALL_PROPERTY macro to allow using a custom property:

Algorithm 92 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.

5.5.3 Create the empty graph with custom edges and vertices

Algorithm 93 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".

5.6 Add a custom edge

Adding a custom edge is very similar to adding a named edge (chapter 3.6).

Algorithm 94 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(const my_edge& v, graph& g) noexcept
{
  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 = 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)').

5.7 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 3.8 , but with our custom edges and vertices intead:

Algorithm 95 Creating K_3 as depicted in figure 11

```
#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 = 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 = 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 51. 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.

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

6.1 Has a my_vertex

Before modifying our vertices, let's first determine if we can find a vertex by its custom type ('my_vertex') in a graph. After obtaing a my_vertex map, we obtain the vertex iterators, dereference these to obtain the vertex descriptors and then compare each vertex its my_vertex with the one desired.

Algorithm 96 Find if there is vertex with a certain my_vertex

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


template <typename graph>
bool has_vertex_with_my_vertex(
  const my_vertex& v,
  const graph& g
) noexcept
{
  const auto my_vertexes_map = get(boost::
      vertex_custom_type, g);

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

This function can be demonstrated as in algorithm 97, where a certain my_vertex cannot be found in an empty graph. After adding the desired my_vertex, it is found.

Algorithm 97 Demonstration of the 'has_vertex_with_my_vertex' function

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

#include "add_custom_vertex.h"
#include "create_empty_custom_vertices_graph.h"
#include "has_vertex_with_my_vertex.h"
#include "install_vertex_custom_type.h"
#include "my_vertex.h"

void has_vertex_with_my_vertex_demo() noexcept
{
  auto g = create_empty_custom_vertices_graph();
  assert(!has_vertex_with_my_vertex(my_vertex("Felix"),g)
     );
  add_custom_vertex(my_vertex("Felix"),g);
  assert(has_vertex_with_my_vertex(my_vertex("Felix"),g))
     ;
}
```

Note that this function only finds if there is at least one vertex with that my_vertex: it does not tell how many vertices with that my_vertex exist in the graph.

6.2 Find a vertex with a certain my_vertex

Where STL functions work with iterators, here we obtain a vertex descriptor (see chapter 2.6) to obtain a handle to the desired vertex. Algorithm 98 shows how to obtain a vertex descriptor to the first vertex found with a specific my_vertex value.

Algorithm 98 Find the first vertex with a certain my_vertex

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

template <typename graph>
typename boost::graph_traits<graph>::vertex_descriptor
find_first_vertex_with_my_vertex(
  const my_vertex& v,
  const graph& g
) noexcept
{
  assert(has_vertex_with_my_vertex(v, g));
  const auto my_vertexes_map = get(boost::
     vertex_custom_type, g);

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

}
```

With the vertex descriptor obtained, one can read and modify the vertex and the edges surrounding it. Algorithm 99 shows some examples of how to do so.

Algorithm 99 Demonstration of the 'find_first_vertex_with_my_vertex' function

```
#include <cassert>

#include "create_custom_vertices_k2_graph.h"
#include "find_first_vertex_with_my_vertex.h"

void find_first_vertex_with_my_vertex_demo() noexcept
{
  const auto g = create_custom_vertices_k2_graph();
  const auto vd = find_first_vertex_with_my_vertex(
    my_vertex("from","source",0.0,0.0),
    g
  );
  assert(boost::out_degree(vd,g) == 1);
  assert(boost::in_degree(vd,g) == 1);
}
```

6.3 Get a vertex its my_vertex

To obtain the name from a vertex descriptor, one needs to pull out the my_vertexes[14] map and then look up the vertex of interest.

Algorithm 100 Get a vertex its my_vertex from its vertex descriptor

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

template <typename graph>
my_vertex get_vertex_my_vertex(
  const typename boost::graph_traits<graph>::
      vertex_descriptor& vd,
  const graph& g
) noexcept
{
  const auto my_vertexes_map = get(boost::
      vertex_custom_type, g);
  return my_vertexes_map[vd];
}
```

[14]Bad English intended: my_vertexes = multiple my_vertex objects, vertices = multiple graph nodes

To use 'get_vertex_my_vertex', one first needs to obtain a vertex descriptor. Algorithm 101 shows a simple example.

Algorithm 101 Demonstration if the 'get_vertex_my_vertex' function

```
#include <cassert>

#include "add_custom_vertex.h"
#include "create_empty_custom_vertices_graph.h"
#include "find_first_vertex_with_my_vertex.h"
#include "get_vertex_my_vertex.h"

void get_vertex_my_vertex_demo() noexcept
{
  auto g = create_empty_custom_vertices_graph();
  const my_vertex name{"Dex"};
  add_custom_vertex(name, g);
  const auto vd = find_first_vertex_with_my_vertex(name,g
     );
  assert(get_vertex_my_vertex(vd,g) == name);
}
```

6.4 Set a vertex its my_vertex

If you know how to get the my_vertex from a vertex descriptor, setting it is just as easy, as shown in algorithm 102.

Algorithm 102 Set a vertex its my_vertex from its vertex descriptor

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

template <typename graph>
void set_vertex_my_vertex(
  const my_vertex& v,
  const typename boost::graph_traits<graph>::
      vertex_descriptor& vd,
  graph& g
) noexcept
{
  const auto my_vertexes_map = get(boost::
      vertex_custom_type, g);
  my_vertexes_map[vd] = v;
}
```

To use 'set_vertex_my_vertex', one first needs to obtain a vertex descriptor. Algorithm 103 shows a simple example.

Algorithm 103 Demonstration if the 'set_vertex_my_vertex' function

```
#include <cassert>

#include "add_custom_vertex.h"
#include "create_empty_custom_vertices_graph.h"
#include "find_first_vertex_with_my_vertex.h"
#include "get_vertex_my_vertex.h"
#include "set_vertex_my_vertex.h"

void set_vertex_my_vertex_demo() noexcept
{
  auto g = create_empty_custom_vertices_graph();
  const my_vertex old_name{"Dex"};
  add_custom_vertex(old_name, g);
  const auto vd = find_first_vertex_with_my_vertex(
      old_name,g);
  assert(get_vertex_my_vertex(vd,g) == old_name);
  const my_vertex new_name{"Diggy"};
  set_vertex_my_vertex(new_name, vd, g);
  assert(get_vertex_my_vertex(vd,g) == new_name);
}
```

6.5 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 104 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
) noexcept
{
  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.

6.6 Has a my_edge

Before modifying our edges, let's first determine if we can find an edge by its custom type ('my_edge') in a graph. After obtaing a my_edge map, we obtain the edge iterators, dereference these to obtain the edge descriptors and then compare each edge its my_edge with the one desired.

Algorithm 105 Find if there is an edge with a certain my_edge

```
#include <boost/graph/properties.hpp>
#include "install_edge_custom_type.h"
#include "my_edge.h"

template <typename graph>
bool has_edge_with_my_edge(
  const my_edge& e,
  const graph& g
) noexcept
{
  const auto my_edges_map = get(boost::edge_custom_type,g
      );

  for (auto p = edges(g);
    p.first != p.second;
    ++p.first) {
    if (get(my_edges_map, *p.first) == e) {
      return true;
    }
  }
  return false;
}
```

This function can be demonstrated as in algorithm 106, where a certain my_edge cannot be found in an empty graph. After adding the desired my_edge, it is found.

Algorithm 106 Demonstration of the 'has_edge_with_my_edge' function

```
#include <cassert>

#include "add_custom_edge.h"
#include "create_empty_custom_edges_and_vertices_graph.h"
#include "has_edge_with_my_edge.h"

void has_edge_with_my_edge_demo() noexcept
{
  auto g = create_empty_custom_edges_and_vertices_graph()
      ;
  assert(!has_edge_with_my_edge(my_edge("Edward"),g));
  add_custom_edge(my_edge("Edward"),g);
  assert(has_edge_with_my_edge(my_edge("Edward"),g));
}
```

Note that this function only finds if there is at least one edge with that my_edge: it does not tell how many edges with that my_edge exist in the graph.

6.7 Find a my_edge

Where STL functions work with iterators, here we obtain an edge descriptor (see chapter 2.12) to obtain a handle to the desired edge. Algorithm 107 shows how to obtain an edge descriptor to the first edge found with a specific my_edge value.

Algorithm 107 Find the first edge with a certain my_edge

```
#include <cassert>
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/properties.hpp>
#include "has_edge_with_my_edge.h"
#include "install_edge_custom_type.h"
#include "my_edge.h"

template <typename graph>
typename boost::graph_traits<graph>::edge_descriptor
find_first_edge_with_my_edge(
  const my_edge& e,
  const graph& g
) noexcept
{
  assert(has_edge_with_my_edge(e, g));
  const auto my_edges_map = get(boost::edge_custom_type,
     g);

  for (auto p = edges(g);
    p.first != p.second;
    ++p.first) {

    if (get(my_edges_map, *p.first) == e) {
      return *p.first;
    }
  }
  return *edges(g).second;
}
```

With the edge descriptor obtained, one can read and modify the edge and the vertices surrounding it. Algorithm 108 shows some examples of how to do so.

Algorithm 108 Demonstration of the 'find_first_edge_with_my_edge' function

```
#include <cassert>

#include "create_custom_edges_and_vertices_k3_graph.h"
#include "find_first_edge_with_my_edge.h"

void find_first_edge_with_my_edge_demo() noexcept
{
  const auto g =
      create_custom_edges_and_vertices_k3_graph();
  const auto ed = find_first_edge_with_my_edge(
    my_edge("AB","first",0.0,0.0),
    g
  );
  assert(boost::source(ed,g) != boost::target(ed,g));
}
```

6.8 Get an edge its my_edge

To obtain the my_edeg from an edge descriptor, one needs to pull out the my_edges map and then look up the my_edge of interest.

Algorithm 109 Get a vertex its my_vertex from its vertex descriptor

```
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/properties.hpp>
#include "install_edge_custom_type.h"
#include "my_edge.h"

template <typename graph>
my_edge get_edge_my_edge(
  const typename boost::graph_traits<graph>::
      edge_descriptor& vd,
  const graph& g
) noexcept
{
  const auto my_edge_map = get(boost::edge_custom_type, g
      );
  return my_edge_map[vd];
}
```

To use 'get_edge_my_edge', one first needs to obtain an edgedescriptor. Algorithm 110 shows a simple example.

Algorithm 110 Demonstration if the 'get_edge_my_edge' function

```
#include <cassert>

#include "add_custom_edge.h"
#include "create_empty_custom_edges_and_vertices_graph.h"
#include "find_first_edge_with_my_edge.h"
#include "get_edge_my_edge.h"

void get_edge_my_edge_demo() noexcept
{
  auto g = create_empty_custom_edges_and_vertices_graph()
      ;
  const my_edge name{"Dex"};
  add_custom_edge(name, g);
  const auto ed = find_first_edge_with_my_edge(name,g);
  assert(get_edge_my_edge(ed,g) == name);
}
```

6.9 Set an edge its my_edge

If you know how to get the my_edge from an edge descriptor, setting it is just as easy, as shown in algorithm 111.

Algorithm 111 Set an edge its my_edge from its edge descriptor

```
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/properties.hpp>
#include "install_edge_custom_type.h"
#include "my_edge.h"

template <typename graph>
void set_edge_my_edge(
  const my_edge& name,
  const typename boost::graph_traits<graph>::
      edge_descriptor& vd,
  graph& g
) noexcept
{
  auto my_edge_map = get(boost::edge_custom_type, g);
  my_edge_map[vd] = name;
}
```

To use 'set_edge_my_edge', one first needs to obtain an edgedescriptor.

Algorithm 112 shows a simple example.

Algorithm 112 Demonstration if the 'set_edge_my_edge' function

```
#include <cassert>

#include "add_custom_edge.h"
#include "create_empty_custom_edges_and_vertices_graph.h"
#include "find_first_edge_with_my_edge.h"
#include "get_edge_my_edge.h"
#include "set_edge_my_edge.h"

void set_edge_my_edge_demo() noexcept
{
  auto g = create_empty_custom_edges_and_vertices_graph()
      ;
  const my_edge old_name{"Dex"};
  add_custom_edge(old_name, g);
  const auto vd = find_first_edge_with_my_edge(old_name,g
      );
  assert(get_edge_my_edge(vd,g) == old_name);
  const my_edge new_name{"Diggy"};
  set_edge_my_edge(new_name, vd, g);
  assert(get_edge_my_edge(vd,g) == new_name);
}
```

6.10 Storing a graph with custom vertices as a .dot

If you used the create_custom_vertices_k2_graph function (algorithm 90) to produce a K_2 graph with vertices associated with my_vertex objects, you can store these my_vertexes additionally with algorithm 113:

Algorithm 113 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++14.

The .dot file created is displayed in algorithm 114:

Algorithm 114 .dot file created from the create_custom_vertices_k2_graph function (algorithm 43)

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

This .dot file corresponds to figure 114:

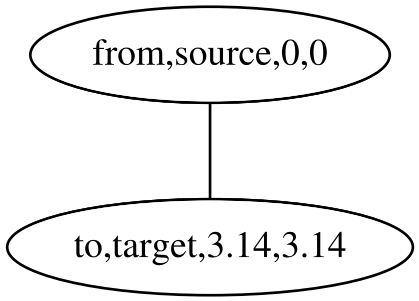

Figure 14: .svg file created from the create_custom_vertices_k2_graph function (algorithm 90) and converted to .svg using the 'convert_dot_to_svg' function (algorithm 124)

6.11 Storing a graph with custom edges and vertices as a .dot

If you used the create_custom_edges_and_vertices_k3_graph function (algorithm 95) to produce a K_3 graph with edges and vertices associated with my_edge and my_vertex objects, you can store these my_edges and my_vertexes additionally with algorithm 115:

Algorithm 115 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"

#error check this with a C++14 compiler

///Save a graph with named vertices to a .dot file
template <typename graph>
void save_custom_edges_and_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++14.

The .dot file created is displayed in algorithm 116:

Algorithm 116 .dot file created from the create_custom_edges_and_vertices_k3_graph function (algorithm 43)

```
graph G {
0[label="top,source,0,0"];
1[label="right,target,3.14,0"];
2[label="left,target,0,3.14"];
0--1 [label="AB,first,0,0"];
1--2 [label="BC,second,3.14,3.14"];
2--0 [label="CA,third,3.14,3.14"];
}
```

This .dot file corresponds to figure 116:

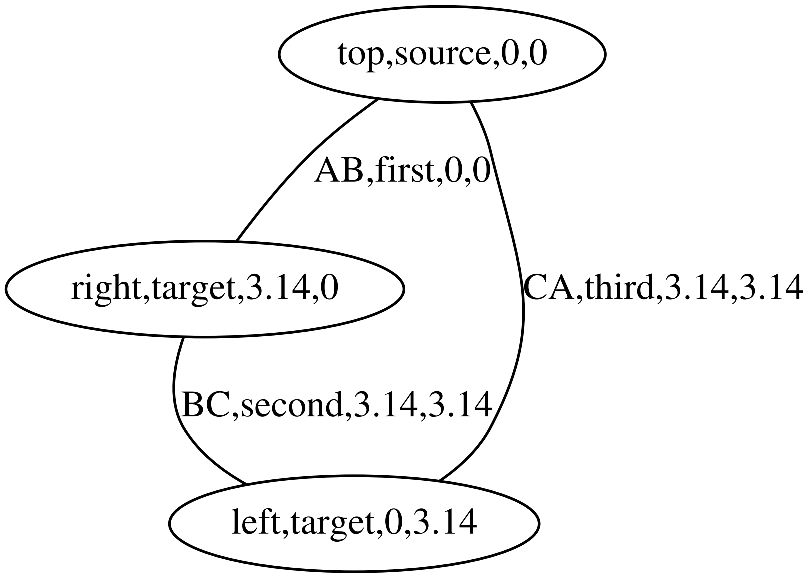

Figure 15: .svg file created from the create_custom_edges_and_vertices_k3_graph function (algorithm 95) and converted to .svg using the 'convert_dot_to_svg' function (algorithm 124)

7 Other graph functions

7.1 Create an empty graph with a graph name property

Algorithm 117 shows the function to create an empty (directed) graph with a graph name.

Algorithm 117 Creating an empty directed graph with a graph name

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

boost::adjacency_list<
  boost::vecS,
  boost::vecS,
  boost::undirectedS,
  boost::no_property,
  boost::no_property,
  boost::property<
    boost::graph_name_t,std::string
  >
>
create_empty_directed_graph_with_graph_name() noexcept
{
  return boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    boost::no_property,
    boost::no_property,
    boost::property<
      boost::graph_name_t,std::string
    >
  >();
}
```

Algorithm 118 demonstrates the 'create_empty_directed_graph_with_graph_name' function.

Algorithm 118 Demonstration of 'create_empty_directed_graph_with_graph_name'

```
#include <cassert>

#include "create_empty_directed_graph_with_graph_name.h"
#include "get_n_edges.h"
#include "get_n_vertices.h"

void create_empty_directed_graph_with_graph_name_demo()
    noexcept
{
  auto g = create_empty_directed_graph_with_graph_name();
  assert(get_n_edges(g) == 0);
  assert(get_n_vertices(g) == 0);
}
```

7.2 Set a graph its name property

If you know, please email me.

Algorithm 119 Set a graph its name

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

template <typename graph>
void set_graph_name(
  const std::string& name,
  graph& g
) noexcept
{
  assert(!"TODO");
  //get(boost::graph_name, g) = name;
  //get(boost::graph_name, g)[g] = name;
  //put(name, boost::graph_name, g);
}
```

Algorithm 120 demonstrates the 'set_graph_name' function.

Algorithm 120 Demonstration of 'set_graph_name'

```
#include <cassert>

#include "create_empty_directed_graph_with_graph_name.h"
#include "get_graph_name.h"
#include "set_graph_name.h"

void set_graph_name_demo() noexcept
{
  assert(!"TODO");
  //No idea
  /*
  auto g = create_empty_directed_graph_with_graph_name();
  boost::edges(
  const std::string old_name{"Gramps"};
  //g[boost::graph_name] = old_name;
  auto m = get(boost::graph_name, g);
  //m[&g] = name;
  //set_graph_name(old_name, g);
  //assert(get_graph_name(g) == old_name);
  */
}
```

7.3 Get a graph its name property

If you know, please email me.

Algorithm 121 Get a graph its name

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

template <typename graph>
std::string get_graph_name(
  const graph& g
) noexcept
{
  return get(boost::graph_name,g);
}
```

Algorithm 122 demonstrates the 'get_graph_name' function.

Algorithm 122 Demonstration of 'get_graph_name'

```
#include <cassert>

#include "create_empty_directed_graph_with_graph_name.h"
#include "get_graph_name.h"
#include "set_graph_name.h"

void get_graph_name_demo() noexcept
{
  assert(!"TODO");
  /*
  auto g = create_empty_directed_graph_with_graph_name();
  const std::string name{"Dex"};
  set_graph_name(name, g);
  assert(get_graph_name(g) == name);
  */
}
```

7.4 Create a K2 graph with a graph name property

If you know, please email me.

7.5 Storing a graph with a graph name property as a .dot

If you know, please email me.

8 Graph gallery

8.1 Markov chain

9 Misc functions

These are some function I needed for creating this tutorial. Although they are not important for working with graphs, I used these heavily. These functions may be compiler-dependent or platform-dependent. I just add them for

9.1 Getting a data type as a std::string

This function will only work under GCC.

Algorithm 123 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() noexcept
{
  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;
}
```

9.2 Convert a .dot to .svg

All illustrations in this tutorial are created by converting .dot to a .svg ('Scalable Vector Graphic') file. This function assumes the program 'dot' is installed, which is part of GraphViz.

Algorithm 124 Convert a .dot file to a .svg

```
#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() noexcept
{
  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;
}
```

'convert_dot_to_svg' makes a system call to the prgram 'dot' to convert the .dot file to an .svg file.

10 Errors

Some common errors.

10.1 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 125:

Algorithm 125 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 125 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.

10.2 No matching function for call to 'clear_out_edges'

This compile-time error occurs when you want to clear the outward edges from a vertex in an undirected graph.

Algorithm 126 Creating the error 'formed reference to void'

```
#include "create_k2_graph.h"

void no_matching_function_for_call_to_clear_out_edges()
    noexcept
{
  auto g = create_k2_graph();
  const auto vd = *boost::vertices(g).first;
  boost::clear_in_edges(vd,g);
}
```

In algorithm 126 an undirected graph is created, a vertex descriptor is obtained, then its out edges are tried to be cleared. Either use a directed graph (which has out edges), or use the 'boost::clear_vertex' function instead.

10.3 No matching function for call to 'clear_in_edges'

See chapter 10.2.

10.4 Undefined reference to boost::detail::graph::read_graphviz_new

You will have to link agains the Boost.Graph and Boost.Regex libraries. In Qt Creator, this is achieved by adding these lines to your Qt Creator project file:

```
LIBS += -lboost_graph -lboost_regex
```

10.5 Property not found: node_id

When loading a graph from file (as in chapter 2.18) you will be using boost::read_graphviz. boost::read_graphviz needs a third argument, of type boost::dynamic_properties. When a graph does not have properties, do not use a default constructed version, but initializate with 'boost::ignore_other_properties' as a constructor argument instead. Algorithm 127 shows how to trigger this run-time error.

Algorithm 127 Storing a graph as a .dot file

```
#include <cassert>
#include <fstream>
#include "is_regular_file.h"
#include "create_empty_undirected_graph.h"
#include "create_k2_graph.h"
#include "save_graph_to_dot.h"

void property_not_found_node_id() noexcept
{
  const std::string dot_filename{"
     property_not_found_node_id.dot"};
  //Create a file
  {
    const auto g = create_k2_graph();
    save_graph_to_dot(g, dot_filename);
    assert(is_regular_file(dot_filename));
  }

  //Try to read that file
  std::ifstream f(dot_filename.c_str());
  auto g = create_empty_undirected_graph();

  //Line below should have been
  //  boost::dynamic_properties p(boost::
     ignore_other_properties);
  boost::dynamic_properties p; //Error

  try {
    boost::read_graphviz(f,g,p);
  }
  catch (std::exception&) {
    return; //Should get here
  }
  assert(!"Should␣not␣get␣here");
}
```

11 Appendix

11.1 List of all edge, graph and vertex properties

The following list is obtained from the file 'boost/graph/properties.hpp'.

Edge	Graph	Vertex
edge_all	graph_all	vertex_all
edge_bundle	graph_bundle	vertex_bundle
edge_capacity	graph_name	vertex_centrality
edge_centrality	graph_visitor	vertex_color
edge_color		vertex_current_degree
edge_discover_time		vertex_degree
edge_finished		vertex_discover_time
edge_flow		vertex_distance
edge_global		vertex_distance2
edge_index		vertex_finish_time
edge_local		vertex_global
edge_local_index		vertex_in_degree
edge_name		vertex_index
edge_owner		vertex_index1
edge_residual_capacity		vertex_index2
edge_reverse		vertex_local
edge_underlying		vertex_local_index
edge_update		vertex_lowpoint
edge_weight		vertex_name
edge_weight2		vertex_out_degree
		vertex_owner
		vertex_potential
		vertex_predecessor
		vertex_priority
		vertex_rank
		vertex_root
		vertex_underlying
		vertex_update

References

[1] Eckel Bruce. Thinking in c++, volume 1. 2002.

[2] Marshall P Cline, Greg Lomow, and Mike Girou. *C++ FAQs*. Pearson Education, 1998.

[3] Jarrod Hollingworth, Bob Swart, and Jamie Allsop. *C++ Builder 5 Developer's Guide with Cdrom*. Sams, 2000.

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

[5] Jesse Liberty. *Sams teach yourself C++ in 24 hours.* Sams Publishing, 2001.

[6] Steve McConnell. *Code complete.* Pearson Education, 2004.

[7] Scott Meyers. *Effective C++: 55 specific ways to improve your programs and designs.* Pearson Education, 2005.

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

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

[10] Bjarne Stroustrup. *The C++ Programming Language (4th edition).* 2013.

[11] Herb Sutter and Andrei Alexandrescu. *C++ coding standards: 101 rules, guidelines, and best practices.* Pearson Education, 2004.

Index