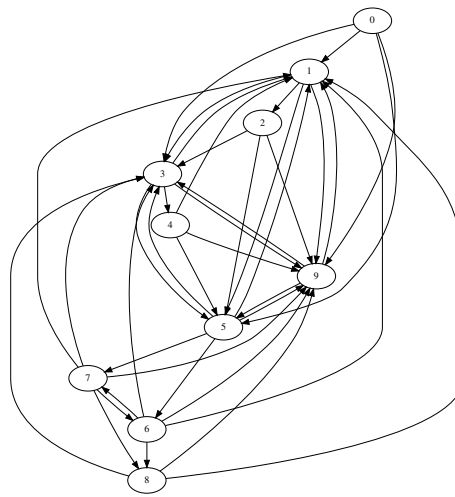


# Boost.Graph Cookbook 1: Basics

Richel Bilderbeek

May 20, 2018



## Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	Why this tutorial . . . . .	5
1.2	Tutorial style . . . . .	6
1.3	Coding style . . . . .	6
1.4	License . . . . .	8
1.5	Feedback . . . . .	8
1.6	Acknowledgements . . . . .	8
1.7	Outline . . . . .	9
<b>2</b>	<b>Building graphs without properties</b>	<b>9</b>
2.1	Creating an empty (directed) graph . . . . .	12
2.2	Creating an empty undirected graph . . . . .	14
2.3	Counting the number of vertices . . . . .	15
2.4	Counting the number of edges . . . . .	16
2.5	Adding a vertex . . . . .	17
2.6	Vertex descriptors . . . . .	18

2.7	Get the vertex iterators . . . . .	19
2.8	Get all vertex descriptors . . . . .	20
2.9	Add an edge . . . . .	21
2.10	boost::add_edge result . . . . .	23
2.11	Getting the edge iterators . . . . .	23
2.12	Edge descriptors . . . . .	24
2.13	Get all edge descriptors . . . . .	25
2.14	Creating a directed graph . . . . .	26
2.14.1	Graph . . . . .	26
2.14.2	Function to create such a graph . . . . .	27
2.14.3	Creating such a graph . . . . .	27
2.14.4	The .dot file produced . . . . .	28
2.14.5	The .svg file produced . . . . .	28
2.15	Creating $K_2$ , a fully connected undirected graph with two vertices	29
2.15.1	Graph . . . . .	29
2.15.2	Function to create such a graph . . . . .	29
2.15.3	Creating such a graph . . . . .	30
2.15.4	The .dot file produced . . . . .	31
2.15.5	The .svg file produced . . . . .	31
2.16	► Creating $K_3$ , a fully connected undirected graph with three vertices . . . . .	32
2.16.1	Graph . . . . .	32
2.16.2	Function to create such a graph . . . . .	32
2.16.3	Creating such a graph . . . . .	33
2.16.4	The .dot file produced . . . . .	33
2.16.5	The .svg file produced . . . . .	34
2.17	► Creating a path graph . . . . .	34
2.17.1	Graph . . . . .	34
2.17.2	Function to create such a graph . . . . .	35
2.17.3	Creating such a graph . . . . .	35
2.17.4	The .dot file produced . . . . .	36
2.17.5	The .svg file produced . . . . .	36
2.18	► Creating a Peterson graph . . . . .	37
2.18.1	Graph . . . . .	37
2.18.2	Function to create such a graph . . . . .	38
2.18.3	Creating such a graph . . . . .	39
2.18.4	The .dot file produced . . . . .	40
2.18.5	The .svg file produced . . . . .	41
<b>3</b>	<b>Working on graphs without properties</b>	<b>42</b>
3.1	Getting the vertices' out degree . . . . .	43
3.2	► Is there an edge between two vertices? . . . . .	44
3.3	► Get the edge between two vertices . . . . .	45
3.4	►► Create a direct-neighbour subgraph from a vertex descriptor	47
3.5	►► Create a direct-neighbour subgraph from a vertex descriptor including inward edges . . . . .	49

3.6	►► Creating all direct-neighbour subgraphs from a graph without properties . . . . .	51
3.7	► Are two graphs isomorphic? . . . . .	52
3.8	►► Count the number of connected components in a directed graph . . . . .	53
3.9	►► Count the number of connected components in an undirected graph . . . . .	55
3.10	►► Count the number of levels in an undirected graph . . . . .	57
3.11	Saving a graph to a .dot file . . . . .	59
3.12	Loading a directed graph from a .dot . . . . .	60
3.13	Loading an undirected graph from a .dot file . . . . .	62
<b>4</b>	<b>Building graphs with bundled vertices</b>	<b>64</b>
4.1	Creating the bundled vertex class . . . . .	65
4.2	Create the empty directed graph with bundled vertices . . . . .	66
4.3	Create the empty undirected graph with bundled vertices . . . . .	67
4.4	Add a bundled vertex . . . . .	67
4.5	Getting the bundled vertices' my_vertexes <sup>1</sup> . . . . .	68
4.6	Creating a two-state Markov chain with bundled vertices . . . . .	68
4.6.1	Graph . . . . .	68
4.6.2	Function to create such a graph . . . . .	69
4.6.3	Creating such a graph . . . . .	70
4.6.4	The .dot file produced . . . . .	71
4.6.5	The .svg file produced . . . . .	73
4.7	Creating $K_2$ with bundled vertices . . . . .	74
4.7.1	Graph . . . . .	74
4.7.2	Function to create such a graph . . . . .	75
4.7.3	Creating such a graph . . . . .	75
4.7.4	The .dot file produced . . . . .	76
4.7.5	The .svg file produced . . . . .	78
<b>5</b>	<b>Working on graphs with bundled vertices</b>	<b>79</b>
5.1	Has a bundled vertex with a my_bundled_vertex . . . . .	79
5.2	Find a bundled vertex with a certain my_bundled_vertex . . . . .	81
5.3	Get a bundled vertex its 'my_bundled_vertex' . . . . .	83
5.4	Set a bundled vertex its my_vertex . . . . .	84
5.5	Setting all bundled vertices' my_vertex objects . . . . .	85
5.6	Storing a graph with bundled vertices as a .dot . . . . .	86
5.7	Loading a directed graph with bundled vertices from a .dot . . . . .	89
5.8	Loading an undirected graph with bundled vertices from a .dot . . . . .	92

---

<sup>1</sup>the name 'my\_vertexes' is chosen to indicate this function returns a container of my\_vertex

<b>6</b>	<b>Building graphs with bundled edges and vertices</b>	<b>94</b>
6.1	Creating the bundled edge class . . . . .	95
6.2	Create an empty directed graph with bundled edges and vertices	97
6.3	Create an empty undirected graph with bundled edges and vertices	98
6.4	Add a bundled edge . . . . .	99
6.5	Getting the bundled edges my_edges . . . . .	101
6.6	Creating a Markov-chain with bundled edges and vertices . . . .	102
6.6.1	Graph . . . . .	102
6.6.2	Function to create such a graph . . . . .	103
6.6.3	Creating such a graph . . . . .	105
6.6.4	The .dot file produced . . . . .	105
6.6.5	The .svg file produced . . . . .	107
6.7	Creating $K_3$ with bundled edges and vertices . . . . .	108
6.7.1	Graph . . . . .	108
6.7.2	Function to create such a graph . . . . .	110
6.7.3	Creating such a graph . . . . .	111
6.7.4	The .dot file produced . . . . .	111
6.7.5	The .svg file produced . . . . .	113
<b>7</b>	<b>Working on graphs with bundled edges and vertices</b>	<b>114</b>
7.1	Has a my_bundled_edge . . . . .	114
7.2	Find a my_bundled_edge . . . . .	115
7.3	Get an edge its my_bundled_edge . . . . .	117
7.4	Set an edge its my_bundled_edge . . . . .	118
7.5	Storing a graph with bundled edges and vertices as a .dot . . . .	120
7.6	Load a directed graph with bundled edges and vertices from a .dot file . . . . .	121
7.7	Load an undirected graph with bundled edges and vertices from a .dot file . . . . .	125
<b>8</b>	<b>Building graphs with a graph name</b>	<b>128</b>
8.1	Create an empty directed graph with a graph name property . .	128
8.2	Create an empty undirected graph with a graph name property .	129
8.3	Get a graph its name property . . . . .	131
8.4	Set a graph its name property . . . . .	132
8.5	Create a directed graph with a graph name property . . . . .	132
8.5.1	Graph . . . . .	132
8.5.2	Function to create such a graph . . . . .	133
8.5.3	Creating such a graph . . . . .	133
8.5.4	The .dot file produced . . . . .	134
8.5.5	The .svg file produced . . . . .	135
8.6	Create an undirected graph with a graph name property . . . .	135
8.6.1	Graph . . . . .	135
8.6.2	Function to create such a graph . . . . .	135
8.6.3	Creating such a graph . . . . .	136
8.6.4	The .dot file produced . . . . .	137

8.6.5	The .svg file produced . . . . .	137
<b>9</b>	<b>Working on graphs with a graph name</b>	<b>137</b>
9.1	Storing a graph with a graph name property as a .dot file . . . .	137
9.2	Loading a directed graph with a graph name property from a .dot file . . . . .	138
9.3	Loading an undirected graph with a graph name property from a .dot file . . . . .	140
<b>10</b>	<b>Other graph functions</b>	<b>142</b>
10.1	Encode a std::string to a Graphviz-friendly format . . . . .	142
10.2	Decode a std::string from a Graphviz-friendly format . . . . .	142
10.3	Check if a std::string is Graphviz-friendly . . . . .	142
<b>11</b>	<b>Misc functions</b>	<b>143</b>
11.1	Getting a data type as a std::string . . . . .	143
11.2	Convert a .dot to .svg . . . . .	144
11.3	Check if a file exists . . . . .	146
<b>12</b>	<b>Errors</b>	<b>146</b>
12.1	Formed reference to void . . . . .	146
12.2	No matching function for call to 'clear_out_edges' . . . . .	147
12.3	No matching function for call to 'clear_in_edges' . . . . .	147
12.4	Undefined reference to boost::detail::graph::read_graphviz_new .	147
12.5	Property not found: node_id . . . . .	147
12.6	Stream zeroes . . . . .	148
<b>13</b>	<b>Appendix</b>	<b>150</b>
13.1	List of all edge, graph and vertex properties . . . . .	150
13.2	Graphviz attributes . . . . .	150

# 1 Introduction

This is 'Boost.Graph Cookbook 1: Basics' (formerly: 'A well-connected C++14 Boost.Graph tutorial'), version 3.0.

## 1.1 Why this tutorial

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.

## 1.2 Tutorial style

**Readable for beginners** This tutorial is aimed at the beginner programmer. This tutorial is intended to take the reader to the level of understanding the book [8] and the Boost.Graph website require. It is about basic graph manipulation, not the more advanced graph algorithms.

**High verbosity** This tutorial is intended to be as verbose, such that a beginner should be able to follow every step, from reading the tutorial from beginning to end chronologically. Especially in the earlier chapters, the rationale behind the code presented is given, including references to the literature. Chapters marked with '►' are optional, less verbose and bring no new information to the storyline.

**Repetitiveness** This tutorial is intended to be as repetitive, such that a beginner can spot the patterns in the code snippets their increasing complexity. Extending code from this tutorial should be as easy as extending the patterns.

**Index** In the index, I did first put all my long-named functions there literally, but this resulted in a very sloppy layout. Instead, the function 'do\_something' can be found as 'Do something' in the index. On the other hand, STL and Boost functions like 'std::do\_something' and 'boost::do\_something' can be found as such in the index.

## 1.3 Coding style

**Concept** For every concept, I will show

- a function that achieves a goal, for example 'create\_empty\_undirected\_graph'
- a test case of that function, that demonstrates how to use the function, for example 'create\_empty\_undirected\_graph\_test'

**C++14** All coding snippets are taken from compiled and tested C++14 code. I chose to use C++14 because it was available to me on all local and remote computers. Next to this, it makes code even shorter than just C++11.

**Coding standard** 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.

**No comments in code** It is important to add comments to code. In this tutorial, however, I have chosen not to put comments in code, as I already describe the function in the tutorial its text. This way, it prevents me from saying the same things twice.

**Trade-off between generic code and readability** It is good to write generic code. In this tutorial, however, I have chosen my functions to have no templated arguments for conciseness and readability. For example, a vertex name is `std::string`, the type for if a vertex is selected is a boolean, and the custom vertex type is of type `'my_custom_vertex'`. I think these choices are reasonable and that the resulting increase in readability is worth it.

**Long function names** 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. Additionally, they explicitly mention their return type (in a simpler way), which may be considered informative.

**Long function names and readability** Due to my long function names and the limitation of  $\approx 50$  characters per line, sometimes the code does get to look a bit awkward. I am sorry for this.

**Use of auto** I prefer to use the keyword `auto` over doubling the lines of code for using statements. Often the 'do' functions return an explicit data type, these can be used for reference. Sometime I deduce the return type using `decltype` and a function with the same return type. When C++17 gets accessible, I will use `'decltype(auto)'`. If you really want to know a type, you can use the `'get_type_name'` function (chapter 11.1).

**Explicit use of namespaces** On the other hand, I am explicit in the namespaces of functions and classes I use, so to distinguish between types like `'std::array'` and `'boost::array'`. Some functions (for example, `'get'`) 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.

**Use of STL algorithms** I try to use STL algorithms wherever I can. Also you should prefer algorithm calls over hand-written for-loops ([9] chapter 18.12.1, [7] item 43). Sometimes using these algorithms becomes a burden on the lines of code. This is because in C++11, a lambda function argument (use by the algorithm) must have its data type specified. It may take multiple lines of 'using' statements being able to do so. In C++14 one can use `'auto'` there as well. So, only if it shortens the number of lines significantly, I use raw for-loops, even though you shouldn't.

**Re-use of functions** The functions I develop in this tutorial are re-used from that moment on. This improves to readability of the code and decreases the number of lines.

**Tested to compile** All functions in this tutorial are tested to compile using Travis CI in both debug and release mode.

**Tested to work** All functions in this tutorial are tested, using the Boost.Test library. Travis CI calls these tests after each push to the repository.

**Availability** The code, as well as this tutorial, can be downloaded from the GitHub at [www.github.com/richelbilderbeek/boost\\_graph\\_cookbook\\_1](http://www.github.com/richelbilderbeek/boost_graph_cookbook_1).

## 1.4 License

This tutorial is licensed under Creative Commons license 4.0. All C++ code is licensed under GPL 3.0.



Figure 1: Creative Commons license 4.0

## 1.5 Feedback

This tutorial is not intended to be perfect yet. For that, I need help and feedback from the community. All referenced feedback is welcome, as well as any constructive 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. Next to this, there are some sections that need to be coded or have its code improved.

## 1.6 Acknowledgements

These are users that improved this tutorial and/or the code behind this tutorial, in chronological order:

- m-dudley, <http://stackoverflow.com/users/111327/m-dudley>
- E. Kawashima
- mat69, <https://www.reddit.com/user/mat69>
- danielhj, <https://www.reddit.com/user/danielhj>
- sehe, <http://stackoverflow.com/users/85371/sehe>



- cv\_and\_me, <http://stackoverflow.com/users/2417774/cv-and-he>
- mywtfmp3

## 1.7 Outline

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.

The distinction between the chapter is in the type of edges and vertices. They can have:

- no properties: see chapter 2
- have a bundled property: see chapter 4

Pivotal chapters are chapters like ‘Finding the first vertex with ...’, as this opens up the door to finding a vertex and manipulating it.

All chapters have a rather similar structure in themselves, as depicted in figure 2.

There are also some bonus chapters, that I have labeled with a ‘►’. These chapters are added I needed these functions myself and adding them would not hurt. Just feel free to skip them, as there will be less theory explained.

## 2 Building graphs 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).

Still, there are two types of graphs that can be constructed: undirected and directed graphs. 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 3. 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 4.

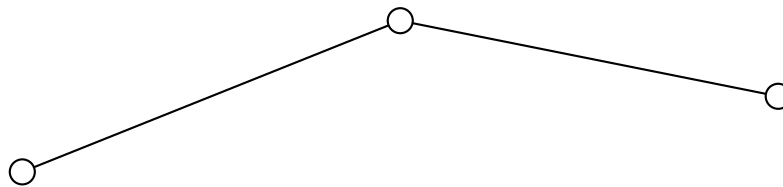


Figure 3: Example of an undirected graph

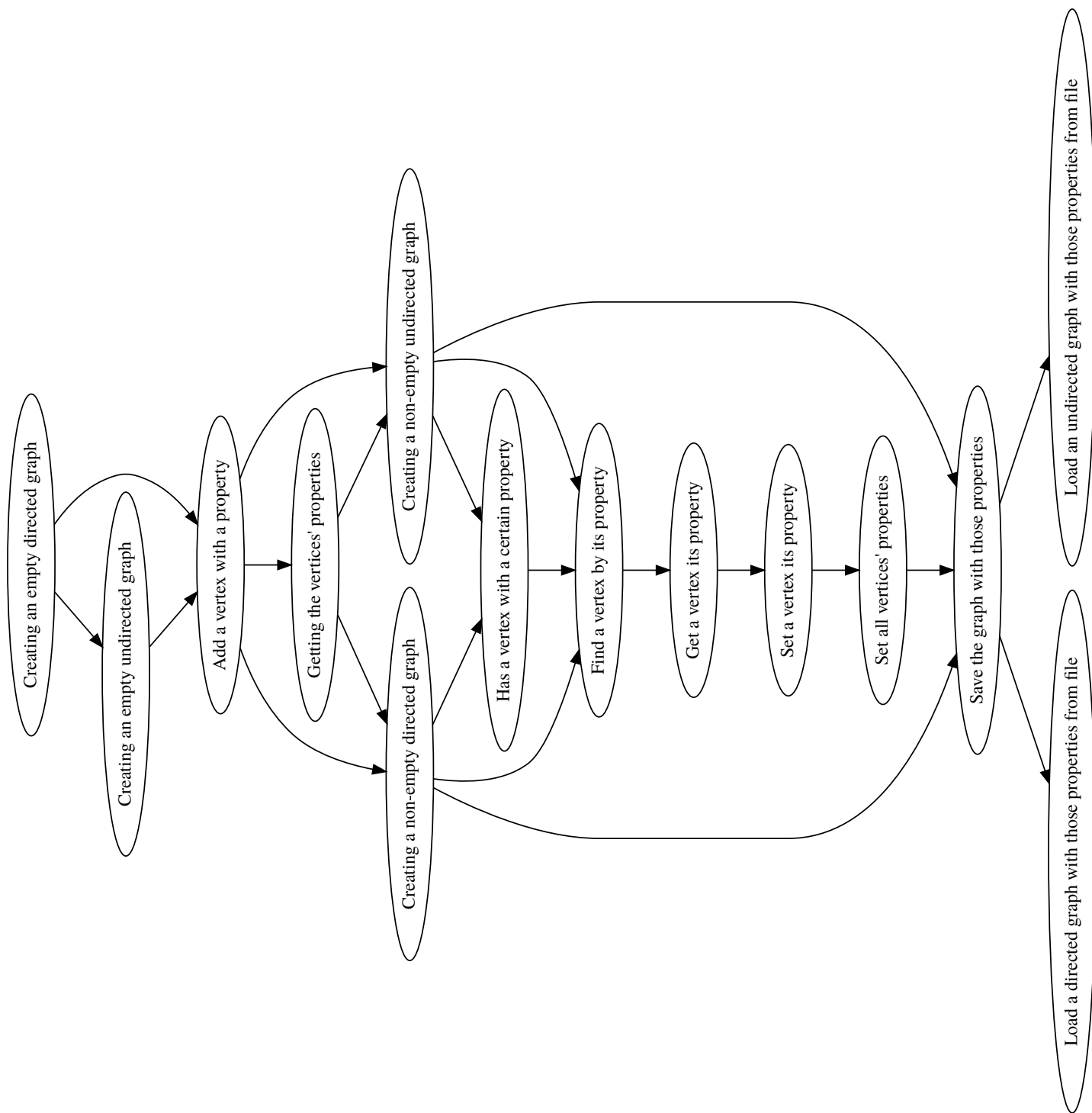


Figure 2: The relations between sub-chapters

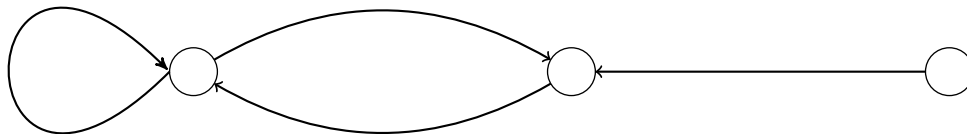


Figure 4: Example of a directed graph

In this chapter, we will build two directed and two undirected graphs:

- An empty (directed) graph, which is the default type: see chapter 2.1
- An empty (undirected) graph: see chapter 2.2
- A two-state Markov chain, a directed graph with two vertices and four edges, chapter 2.14
- $K_2$ , an undirected graph with two vertices and one edge, chapter 2.15

Creating an empty graph may sound trivial, it is not, thanks to the versatility of the Boost.Graph library.

In the process of creating graphs, some basic (sometimes bordering trivial) functions are encountered:

- 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

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

After this chapter you may want to:

- Building graphs with named vertices: see chapter ??
- Building graphs with bundled vertices: see chapter 4
- Building graphs with custom vertices: see chapter ??
- Building graphs with a graph name: see chapter 8

## 2.1 Creating an empty (directed) graph

Let's create an empty graph!

Algorithm 1 shows the function to create an empty graph.

---

**Algorithm 1** Creating an empty (directed) graph

---

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

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

---

The code consists out of an `#include` and a function definition. The `#include` tells the compiler to read the header file `'adjacency_list.hpp'`. A header file (often with a `'.h'` or `'.hpp'` extension) contains class and functions declarations and/or definitions. The header file `'adjacency_list.hpp'` contains the `boost::adjacency_list` class definition. Without including this file, you will get compile errors like `'definition of boost::adjacency_list unknown'`<sup>2</sup>. 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<sup>3</sup>, so it is preferred to mark it `noexcept` ([10] chapter 13.7).

---

<sup>2</sup>In practice, these compiler error messages will be longer, bordering the unreadable

<sup>3</sup>if the function would throw because it cannot allocate this little piece of memory, you are already in big trouble

- a function body: all the function body does is implicitly create its return type by using the ‘{}’. An alternative syntax would be ‘return boost::adjacency\_list<>()’, which is needlessly longer

Algorithm 2 demonstrates the ‘create\_empty\_directed\_graph’ function. This demonstration is embedded within a Boost.Test unit test case. It includes a Boost.Test header to allow to use the Boost.Test framework. Additionally, a header file is included with the same name as the function<sup>4</sup>. This allows use to be able to use the function. The test case creates an empty graph and stores it. Instead of specifying the data type explicitly, ‘auto’ is used (this is preferred, [10] chapter 31.6), which lets the compiler figure out the type itself.

---

**Algorithm 2** Demonstration of ‘create\_empty\_directed\_graph’

---

```
#include <boost/test/unit_test.hpp>
#include "create_empty_directed_graph.h"

BOOST_AUTO_TEST_CASE(test_create_empty_directed_graph)
{
    const auto g = create_empty_directed_graph();
    BOOST_CHECK(boost::is_directed(g));
}
```

---

Congratulations, you’ve just created a boost::adjacency\_list with its default template arguments. The boost::adjacency\_list is the most commonly used graph type, the other is the boost::adjacency\_matrix. We do not do anything with it yet, but still, you’ve just created a graph, in which:

- The out edges and 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

It stores its edges, out edges and vertices in a two different STL<sup>5</sup> 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).

---

<sup>4</sup>I do not think it is important to have creative names

<sup>5</sup>Standard Template Library, the standard library

## 2.2 Creating an empty undirected graph

Let's create another empty graph! This time, we even make it undirected!

Algorithm 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 {};
}
```

---

This algorithm differs from the ‘create\_empty\_directed\_graph’ function (algorithm 1) in that there are three template arguments that need to be specified in the creation of the boost::adjacency\_list:

- the first ‘boost::vecS’: select (that is what the ‘S’ means) that out edges are stored in a std::vector. This is the default way.
- the second ‘boost::vecS’: select that the graph vertices are stored in a std::vector. This is the default way.
- ‘boost::undirectedS’: select that the graph is undirected. This is all we needed to change. By default, this argument is boost::directed

Algorithm 4 demonstrates the ‘create\_empty\_undirected\_graph’ function.

---

**Algorithm 4** Demonstration of ‘create\_empty\_undirected\_graph’

---

```
#include <boost/test/unit_test.hpp>
#include "create_empty_undirected_graph.h"

BOOST_AUTO_TEST_CASE(test_create_empty_undirected_graph)
{
    const auto g = create_empty_undirected_graph();
    BOOST_CHECK(boost::is_undirected(g));
}
```

---

Congratulations, with algorithm 4, you’ve just created an undirected graph in which:

- The out edges and vertices are stored in a `std::vector`
- The graph is undirected
- Vertices, edges and graph have no properties
- Edges are stored in a `std::list`

## 2.3 Counting the number of vertices

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

---

**Algorithm 5** Count the number of vertices

---

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

template <typename graph>
int get_n_vertices(const graph& g) noexcept
{
    const int n{
        static_cast<int>(boost::num_vertices(g))
    };
    assert(static_cast<unsigned long>(n)
        == boost::num_vertices(g)
    );
    return n;
}
```

---

The function ‘`get_n_vertices`’ takes the result of `boost::num_vertices`, converts it to `int` and checks if there was conversion error. We do so, as one should prefer using signed data types over unsigned ones in an interface ([4] chapter 9.2.2). To do so, in the function body its first statement, the unsigned long produced by `boost::num_vertices` get converted to an `int` using a `static_cast`. Using an unsigned integer over a (signed) integer for the sake of gaining that one more bit ([9] chapter 4.4) 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` checks if the conversion back to unsigned long re-creates the original value, to check if no information has been lost. If information is lost, 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 <boost/test/unit_test.hpp>
#include "create_empty_directed_graph.h"
#include "create_empty_undirected_graph.h"
#include "get_n_vertices.h"

BOOST_AUTO_TEST_CASE(test_get_n_vertices)
{
    const auto g = create_empty_directed_graph();
    BOOST_CHECK(get_n_vertices(g) == 0);

    const auto h = create_empty_undirected_graph();
    BOOST_CHECK(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:

---

**Algorithm 7** Count the number of edges

---

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

template <typename graph>
int get_n_edges(const graph& g) noexcept
{
    const int n{
        static_cast<int>(boost::num_edges(g))
    };
    assert(static_cast<unsigned long>(n)
        == boost::num_edges(g)
    );
    return n;
}
```

---

This code is similar to the ‘get\_n\_vertices’ function (algorithm 5, see rationale there) except ‘`boost::num_edges`’ is used, instead of ‘`boost::num_vertices`’,



which also returns an unsigned long.

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 <boost/test/unit_test.hpp>
#include "create_empty_directed_graph.h"
#include "create_empty_undirected_graph.h"
#include "get_n_edges.h"

BOOST_AUTO_TEST_CASE(test_get_n_edges)
{
    const auto g = create_empty_directed_graph();
    BOOST_CHECK(get_n_edges(g) == 0);

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

---

## 2.5 Adding 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 <type_traits>
#include <boost/graph/adjacency_list.hpp>

template <typename graph>
typename boost::graph_traits<graph>::vertex_descriptor
add_vertex(graph& g) noexcept
{
    static_assert(!std::is_const<graph>::value,
        "graph_cannot_be_const");
    const auto vd = boost::add_vertex(g);
    return vd;
}
```

---

The static\_assert at the top of the function checks during compiling if the function is called with a non-const graph. One can freely omit this static\_assert: you will get a compiler error anyways, be it a less helpful one.

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. To allow for this already, ‘`add_vertex`’ also returns a vertex descriptor.

Algorithm 10 shows how to add a vertex to a directed and undirected graph.

---

**Algorithm 10** Demonstration of the ‘`add_vertex`’ function

---

```
#include <boost/test/unit_test.hpp>
#include "add_vertex.h"
#include "create_empty_directed_graph.h"
#include "create_empty_undirected_graph.h"

BOOST_AUTO_TEST_CASE(test_add_vertex)
{
    auto g = create_empty_undirected_graph();
    add_vertex(g);
    BOOST_CHECK(boost::num_vertices(g) == 1);

    auto h = create_empty_directed_graph();
    add_vertex(h);
    BOOST_CHECK(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 dereferencing a vertex iterator (see chapter 2.8). To do so, we first obtain some vertex iterators in chapter 2.7).

Vertex descriptors are used to:

- add an edge between two vertices, see chapter 2.9
- obtain properties of vertex a vertex, for example the vertex its out degrees (chapter 3.1), the vertex its name (chapter ??), or a custom vertex property (chapter ??)

In this tutorial, vertex descriptors have named prefixed with ‘`vd_`’, for example ‘`vd_1`’.

## 2.7 Get the vertex iterators

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 through these steps:

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

‘vertices’ (not ‘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 (its descriptor, to be precise). 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 <typename graph>
std::pair<
    typename graph::vertex_iterator,
    typename graph::vertex_iterator
>
get_vertex_iterators(const graph& g) noexcept
{
    return vertices(g);
}
```

---

This is a somewhat trivial function, as it forwards the function call to ‘vertices’ (not ‘boost::vertices’).

These vertex iterators can be dereferenced to obtain the vertex descriptors. Note that ‘get\_vertex\_iterators’ will not be used often in isolation: usually one obtains the vertex descriptors immediately. Just for your reference, 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\_vertex\_iterators’

---

```
#include <boost/test/unit_test.hpp>
#include "create_empty_directed_graph.h"
#include "create_empty_undirected_graph.h"
#include "get_vertex_iterators.h"

BOOST_AUTO_TEST_CASE(test_get_vertex_iterators)
{
    const auto g = create_empty_undirected_graph();
    const auto vip_g = get_vertex_iterators(g);
    BOOST_CHECK(vip_g.first == vip_g.second);

    const auto h = create_empty_directed_graph();
    const auto vip_h = get_vertex_iterators(h);
    BOOST_CHECK(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/adjacency_list.hpp>
#include <boost/graph/graph_traits.hpp>

template <typename graph>
std::vector<
    typename boost::graph_traits<graph>::vertex_descriptor
>
get_vertex_descriptors(const graph& g) noexcept
{
    using vd = typename graph::vertex_descriptor;

    std::vector<vd> vds(boost::num_vertices(g));
    const auto vis = vertices(g);
    std::copy(vis.first, vis.second, std::begin(vds));
    return vds;
}
```

---

This is the first more complex piece of code. In the first lines, some ‘using’ statements allow for shorter type names<sup>6</sup>.

The `std::vector` to serve as a return value is created at the needed size, which is the number of vertices.

The function ‘vertices’ (not `boost::vertices`!) returns a vertex iterator pair. These iterators are used by `std::copy` to iterate over. `std::copy` is an STL algorithm to copy a half-open range. Prefer algorithm calls over hand-written for-loops ([9] chapter 18.12.1, [7] item 43).

In this case, we copy all vertex descriptors in the range produced by ‘vertices’ to the `std::vector`.

This function will not be used in practice: one iterates over the vertices directly instead, saving the cost of creating a `std::vector`. This function is only shown as an illustration.

Algorithm 14 demonstrates that an empty graph has no vertex descriptors:

---

**Algorithm 14** Demonstration of ‘get\_vertex\_descriptors’

---

```
#include <boost/test/unit_test.hpp>
#include "create_empty_directed_graph.h"
#include "create_empty_undirected_graph.h"
#include "get_vertex_descriptors.h"

BOOST_AUTO_TEST_CASE(test_get_vertex_descriptors)
{
    const auto g = create_empty_undirected_graph();
    const auto vds_g = get_vertex_descriptors(g);
    BOOST_CHECK(vds_g.empty());

    const auto h = create_empty_directed_graph();
    const auto vds_h = get_vertex_descriptors(h);
    BOOST_CHECK(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 within a 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`:

---

<sup>6</sup>which may be necessary just to create a tutorial with code snippets that are readable

---

**Algorithm 15** Adding (two vertices and) an edge to a graph

---

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

template <typename graph>
typename boost::graph_traits<graph>::edge_descriptor
add_edge(graph& g) noexcept
{
    static_assert(!std::is_const<graph>::value,
        "graph_cannot_be_const");
    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 aer.first;
}
```

---

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 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 successful. Insertion can fail if an edge is already present and duplicates are not allowed.

A demonstration of add\_edge is shown in algorithm 16, in which an edge is added to both a directed and undirected graph, after which the number of edges and vertices is checked.

---

**Algorithm 16** Demonstration of ‘add\_edge’

---

```
#include <boost/test/unit_test.hpp>
#include "add_edge.h"
#include "create_empty_directed_graph.h"
#include "create_empty_undirected_graph.h"

BOOST_AUTO_TEST_CASE(test_add_edge)
{
    auto g = create_empty_undirected_graph();
    add_edge(g);
    BOOST_CHECK(boost::num_vertices(g) == 2);
    BOOST_CHECK(boost::num_edges(g) == 1);

    auto h = create_empty_directed_graph();
    add_edge(h);
    BOOST_CHECK(boost::num_vertices(h) == 2);
    BOOST_CHECK(boost::num_edges(h) == 1);
}
```

---

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, which indicates insertion success.

In this tutorial, boost::add\_edge results have named prefixed with ‘aer\_’, for example ‘aer\_1’.

## 2.11 Getting the edge iterators

You cannot get the edges directly. Instead, working on the edges of a graph is done through these steps:

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

‘edges’ (not 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’. Algorithm 17 shows how to obtain these:

---

**Algorithm 17** Get the edge iterators of a graph

---

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

template <typename graph>
std::pair<
    typename graph::edge_iterator,
    typename graph::edge_iterator
>
get_edge_iterators(const graph& g) noexcept
{
    return 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 immediately.

Algorithm 18 demonstrates ‘get\_edge\_iterators’ 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\_edge\_iterators’

---

```
#include <boost/test/unit_test.hpp>
#include "create_empty_directed_graph.h"
#include "create_empty_undirected_graph.h"
#include "get_edge_iterators.h"

BOOST_AUTO_TEST_CASE(test_get_edge_iterators)
{
    const auto g = create_empty_undirected_graph();
    const auto eip_g = get_edge_iterators(g);
    BOOST_CHECK(eip_g.first == eip_g.second);

    auto h = create_empty_directed_graph();
    const auto eip_h = get_edge_iterators(h);
    BOOST_CHECK(eip_h.first == eip_h.second);
}
```

---

## 2.12 Edge descriptors

An edge descriptor is a handle to an edge within a graph. They are similar to vertex descriptors (chapter 2.6).



Edge descriptors are used to obtain the name, or other properties, of an edge. In this tutorial, edge descriptors have names prefixed with ‘ed\_’, for example ‘ed\_1’.

### 2.13 Get all edge descriptors

Obtaining all edge descriptors is similar to obtaining all vertex descriptors (algorithm 13), as shown in algorithm 19:

---

**Algorithm 19** Get all edge descriptors of a graph

---

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

template <typename 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> v(boost::num_edges(g));
    const auto eip = edges(g);
    std::copy(eip.first, eip.second, std::begin(v));
    return v;
}
```

---

The only difference is that instead of the function ‘vertices’ (not `boost::vertices!`), ‘edges’ (not `boost::edges!`) is used.

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 <boost/test/unit_test.hpp>
#include "create_empty_directed_graph.h"
#include "create_empty_undirected_graph.h"
#include "get_edge_descriptors.h"

BOOST_AUTO_TEST_CASE(test_get_edge_descriptors)
{
    const auto g = create_empty_directed_graph();
    const auto eds_g = get_edge_descriptors(g);
    BOOST_CHECK(eds_g.empty());

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

---

## 2.14 Creating a directed graph

Finally, we are going to create a directed non-empty graph!

### 2.14.1 Graph

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

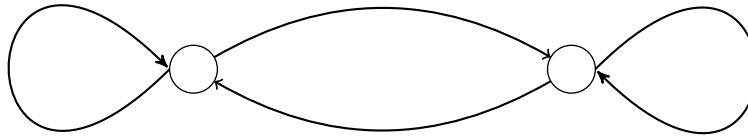


Figure 5: The two-state Markov chain

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

### 2.14.2 Function to create such a graph

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

---

**Algorithm 21** Creating the two-state Markov chain as depicted in figure 5

---

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

boost::adjacency_list<
create_markov_chain() noexcept
{
    auto g = create_empty_directed_graph();
    const auto vd_a = boost::add_vertex(g);
    const auto vd_b = boost::add_vertex(g);
    boost::add_edge(vd_a, vd_a, g);
    boost::add_edge(vd_a, vd_b, g);
    boost::add_edge(vd_b, vd_a, g);
    boost::add_edge(vd_b, vd_b, g);
    return g;
}
```

---

Instead of typing the complete type, we call the ‘create\_empty\_directed\_graph’ function, and let auto figure out the type. The vertex descriptors (see chapter 2.6) created by two boost::add\_vertex calls are stored to add an edge to the graph. Then boost::add\_edge is called four times. Every time, its return type (see chapter 2.10) is checked for a successful insertion.

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

### 2.14.3 Creating such a graph

Algorithm 22 demonstrates the ‘create\_markov\_chain\_graph’ function and checks if it has the correct amount of edges and vertices:

---

**Algorithm 22** Demonstration of the ‘create\_markov\_chain’

---

```
#include <boost/test/unit_test.hpp>
#include "create_markov_chain.h"

BOOST_AUTO_TEST_CASE(test_create_markov_chain)
{
    const auto g = create_markov_chain();
    BOOST_CHECK(boost::num_vertices(g) == 2);
    BOOST_CHECK(boost::num_edges(g) == 4);
}
```

---

#### 2.14.4 The .dot file produced

Running a bit ahead, this graph can be converted to a .dot file using the ‘save\_graph\_to\_dot’ function (algorithm 55). The .dot file created is displayed in algorithm 23:

---

**Algorithm 23** .dot file created from the ‘create\_markov\_chain\_graph’ function (algorithm 21), converted from graph to .dot file using algorithm 55

---

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

---

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

- The first word, ‘digraph’, denotes a directed graph (where ‘graph’ would have indicated an undirected graph)
- The edges are written as ‘->’ (where undirected connections would be written as ‘-’)

#### 2.14.5 The .svg file produced

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

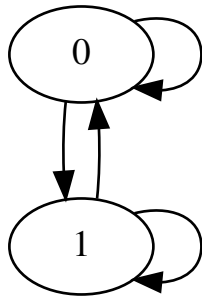


Figure 6: .svg file created from the ‘create\_markov\_chain’ function (algorithm 21) its .dot file and converted from .dot file to .svg using algorithm 136

This figure shows that the graph is directed, as the edges have arrow heads. The vertices display the node index, which is the default behavior.

## 2.15 Creating $K_2$ , a fully connected undirected graph with two vertices

Finally, we are going to create an undirected non-empty graph!

### 2.15.1 Graph

To create a fully connected undirected graph with two vertices (also called  $K_2$ ), one needs two vertices and one (undirected) edge, as depicted in figure 7.



Figure 7:  $K_2$ : a fully connected undirected graph with two vertices

### 2.15.2 Function to create such a graph

To create  $K_2$ , the following code can be used:

---

**Algorithm 24** Creating  $K_2$  as depicted in figure 7

---

```
#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);
    boost::add_edge(vd_a, vd_b, g);
    return g;
}
```

---

This code is very similar to the ‘add\_edge’ function (algorithm 15). Instead of typing the graph its type, we call the ‘create\_empty\_undirected\_graph’ function and let auto figure it out. 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 successful.

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

### 2.15.3 Creating such a graph

Algorithm 25 demonstrates how to ‘create\_k2\_graph’ and checks if it has the correct amount of edges and vertices:

---

**Algorithm 25** Demonstration of ‘create\_k2\_graph’

---

```
#include <boost/test/unit_test.hpp>
#include "create_k2_graph.h"

BOOST_AUTO_TEST_CASE(test_create_k2_graph)
{
    const auto g = create_k2_graph();
    BOOST_CHECK(boost::num_vertices(g) == 2);
    BOOST_CHECK(boost::num_edges(g) == 1);
}
```

---

#### 2.15.4 The .dot file produced

Running a bit ahead, this graph can be converted to the .dot file as shown in algorithm 26:

---

**Algorithm 26** .dot file created from the ‘create\_k2\_graph’ function (algorithm 24), converted from graph to .dot file using algorithm 55

---

```
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 ‘<>’)

#### 2.15.5 The .svg file produced

Continuing to running a bit ahead, this .dot file can be converted to the .svg as shown in figure 8:

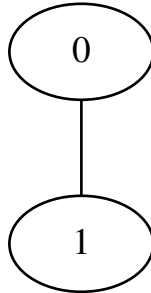


Figure 8: .svg file created from the ‘create\_k2\_graph’ function (algorithm 24) its .dot file, converted from .dot file to .svg using algorithm 136

Also this figure shows that the graph is undirected, otherwise the edge would have one or two arrow heads. The vertices display the node index, which is the default behavior.

## 2.16 ► Creating $K_3$ , a fully connected undirected graph with three vertices

This is an extension of the previous chapter

### 2.16.1 Graph

To create a fully connected undirected graph with two vertices (also called  $K_2$ ), one needs two vertices and one (undirected) edge, as depicted in figure 9.

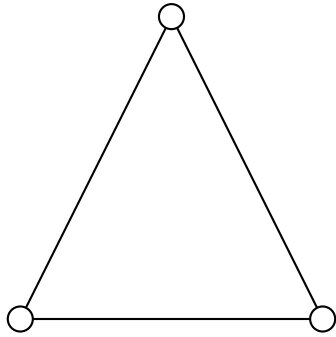


Figure 9:  $K_3$ : a fully connected graph with three edges and vertices

### 2.16.2 Function to create such a graph

To create  $K_3$ , the following code can be used:



---

**Algorithm 27** Creating  $K_3$  as depicted in figure 9

---

```
#include <cassert>
#include "create_empty_undirected_graph.h"
#include "create_k3_graph.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS
>
create_k3_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 vd_c = boost::add_vertex(g);
    boost::add_edge(vd_a, vd_b, g);
    boost::add_edge(vd_b, vd_c, g);
    boost::add_edge(vd_c, vd_a, g);
    return g;
}
```

---

### 2.16.3 Creating such a graph

Algorithm 28 demonstrates how to ‘create\_k2\_graph’ and checks if it has the correct amount of edges and vertices:

---

**Algorithm 28** Demonstration of ‘create\_k3\_graph’

---

```
#include <boost/test/unit_test.hpp>
#include "create_k3_graph.h"

BOOST_AUTO_TEST_CASE(test_create_k3_graph)
{
    const auto g = create_k3_graph();
    BOOST_CHECK(boost::num_edges(g) == 3);
    BOOST_CHECK(boost::num_vertices(g) == 3);
}
```

---

### 2.16.4 The .dot file produced

This graph can be converted to the .dot file as shown in algorithm 29:

---

**Algorithm 29** .dot file created from the ‘create\_k3\_graph’ function (algorithm 27), converted from graph to .dot file using algorithm 55

---

```
graph G {  
0;  
1;  
2;  
0--1 ;  
1--2 ;  
2--0 ;  
}
```

---

### 2.16.5 The .svg file produced

Continuing to running a bit ahead, this .dot file can be converted to the .svg as shown in figure 10:

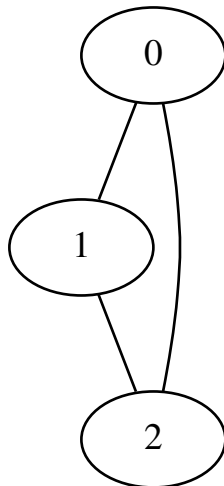


Figure 10: .svg file created from the ‘create\_k3\_graph’ function (algorithm 27) its .dot file, converted from .dot file to .svg using algorithm 136

## 2.17 ► Creating a path graph

A path graph is a linear graph without any branches

### 2.17.1 Graph

Here I show a path graph with four vertices (see figure 11):

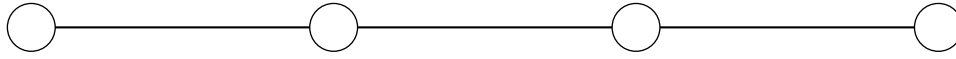


Figure 11: A path graph with four vertices

### 2.17.2 Function to create such a graph

To create a path graph, the following code can be used:

---

**Algorithm 30** Creating a path graph as depicted in figure 11

---

```
#include "create_empty_undirected_graph.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS
>
create_path_graph(const size_t n_vertices) noexcept
{
    auto g = create_empty_undirected_graph();
    if (n_vertices == 0) return g;
    auto vd_1 = boost::add_vertex(g);
    if (n_vertices == 1) return g;
    for (size_t i=1; i!=n_vertices; ++i)
    {
        auto vd_2 = boost::add_vertex(g);
        boost::add_edge(vd_1, vd_2, g);
        vd_1 = vd_2;
    }
    return g;
}
```

---

### 2.17.3 Creating such a graph

Algorithm 31 demonstrates how to ‘create\_k2\_graph’ and checks if it has the correct amount of edges and vertices:

---

**Algorithm 31** Demonstration of ‘create\_path\_graph’

---

```
#include <boost/test/unit_test.hpp>
#include "create_path_graph.h"

BOOST_AUTO_TEST_CASE(test_create_path_graph)
{
    const auto g = create_path_graph(4);
    BOOST_CHECK(boost::num_edges(g) == 3);
    BOOST_CHECK(boost::num_vertices(g) == 4);
}
```

---

#### 2.17.4 The .dot file produced

This graph can be converted to the .dot file as shown in algorithm 32:

---

**Algorithm 32** .dot file created from the ‘create\_path\_graph’ function (algorithm 30), converted from graph to .dot file using algorithm 55

---

```
graph G {
0;
1;
2;
3;
0--1 ;
1--2 ;
2--3 ;
}
```

---

#### 2.17.5 The .svg file produced

The .dot file can be converted to the .svg as shown in figure 12:

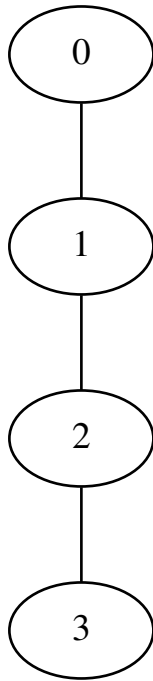


Figure 12: .svg file created from the ‘create\_path\_graph’ function (algorithm 30) its .dot file, converted from .dot file to .svg using algorithm 136

## 2.18 ► Creating a Peterson graph

A Petersen graph is the first graph with interesting properties.

### 2.18.1 Graph

To create a Petersen graph, one needs five vertices and five undirected edges, as depicted in figure 13.

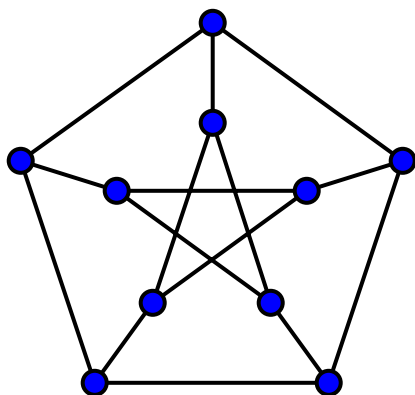


Figure 13: A Petersen graph (from [https://en.wikipedia.org/wiki/Petersen\\_graph](https://en.wikipedia.org/wiki/Petersen_graph))

### 2.18.2 Function to create such a graph

To create a Petersen graph, the following code can be used:

---

**Algorithm 33** Creating Petersen graph as depicted in figure 13

---

```
#include <cassert>
#include <vector>
#include "create_empty_undirected_graph.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS
>
create_petersen_graph() noexcept
{
    using vd = decltype(create_empty_undirected_graph())::
        vertex_descriptor;

    auto g = create_empty_undirected_graph();

    std::vector<vd> v; //Outer
    for (int i=0; i!=5; ++i) {
        v.push_back(boost::add_vertex(g));
    }
    std::vector<vd> w; //Inner
    for (int i=0; i!=5; ++i) {
        w.push_back(boost::add_vertex(g));
    }
    //Outer ring
    for (int i=0; i!=5; ++i) {
        boost::add_edge(v[i], v[(i + 1) % 5], g);
    }
    //Spoke
    for (int i=0; i!=5; ++i) {
        boost::add_edge(v[i], w[i], g);
    }
    //Inner pentagram
    for (int i=0; i!=5; ++i) {
        boost::add_edge(w[i], w[(i + 2) % 5], g);
    }
    return g;
}
```

---

### 2.18.3 Creating such a graph

Algorithm 34 demonstrates how to use ‘create\_petersen\_graph’ and checks if it has the correct amount of edges and vertices:

---

**Algorithm 34** Demonstration of ‘create\_k3\_graph’

---

```
#include <boost/test/unit_test.hpp>
#include "create_petersen_graph.h"

BOOST_AUTO_TEST_CASE(test_create_petersen_graph)
{
    const auto g = create_petersen_graph();
    BOOST_CHECK(boost::num_edges(g) == 15);
    BOOST_CHECK(boost::num_vertices(g) == 10);
}
```

---

#### 2.18.4 The .dot file produced

This graph can be converted to the .dot file as shown in algorithm 35:



---

**Algorithm 35** .dot file created from the ‘create\_petersen\_graph’ function (algorithm 33), converted from graph to .dot file using algorithm 55

---

```
graph G {  
0;  
1;  
2;  
3;  
4;  
5;  
6;  
7;  
8;  
9;  
0--1 ;  
1--2 ;  
2--3 ;  
3--4 ;  
4--0 ;  
0--5 ;  
1--6 ;  
2--7 ;  
3--8 ;  
4--9 ;  
5--7 ;  
6--8 ;  
7--9 ;  
8--5 ;  
9--6 ;  
}
```

---

#### 2.18.5 The .svg file produced

This .dot file can be converted to the .svg as shown in figure 14:

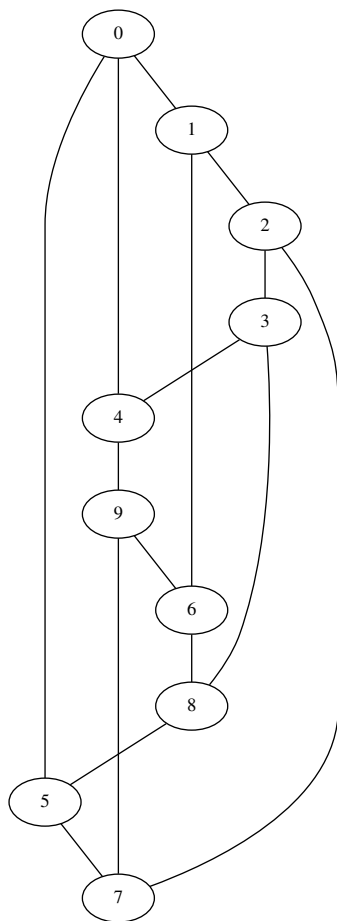


Figure 14: .svg file created from the ‘create\_petersen\_graph’ function (algorithm 33) its .dot file, converted from .dot file to .svg using algorithm 136

### 3 Working on graphs without properties

Now that we can build a graph, there are some things we can do.

- Getting the vertices’ out degrees: see chapter 3.1
- Create a direct-neighbour subgraph from a vertex descriptor
- Create all direct-neighbour subgraphs from a graphs
- Saving a graph without properties to .dot file: see chapter 3.11
- Loading an undirected graph without properties from .dot file: see chapter 3.13

- Loading a directed graph without properties from .dot file: see chapter 3.12

### 3.1 Getting the vertices' out degree

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.

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 'in\_degree' (not 'boost::in\_degree')
- out degree: the number of outgoing connections, using 'out\_degree' (not 'boost::out\_degree')
- degree: sum of the in degree and out degree, using 'degree' (not 'boost::degree')

Algorithm 36 shows how to obtain these:

---

**Algorithm 36** Get the vertices' out degrees

---

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

template <typename graph>
std::vector<int> get_vertex_out_degrees(
    const graph& g
) noexcept
{
    using vd = typename graph::vertex_descriptor;

    std::vector<int> v(boost::num_vertices(g));
    const auto vip = vertices(g);
    std::transform(vip.first, vip.second, std::begin(v),
        [&g](const vd& d) {
            return out_degree(d,g);
        }
    );
    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 that the  $K_2$  graph and the two-state Markov chain are rather simple, we can use it to demonstrate ‘get\_vertex\_out\_degrees’ on, as shown in algorithm 37.

---

**Algorithm 37** Demonstration of the ‘get\_vertex\_out\_degrees’ function

---

```
#include <boost/test/unit_test.hpp>
#include "create_k2_graph.h"
#include "create_markov_chain.h"
#include "get_vertex_out_degrees.h"

BOOST_AUTO_TEST_CASE(test_get_vertex_out_degrees)
{
    const auto g = create_k2_graph();
    const std::vector<int> expected_out_degrees_g{1,1};
    const std::vector<int> vertex_out_degrees_g{
        get_vertex_out_degrees(g)
    };
    BOOST_CHECK(expected_out_degrees_g
        == vertex_out_degrees_g
    );

    const auto h = create_markov_chain();
    const std::vector<int> expected_out_degrees_h{2,2};
    const std::vector<int> vertex_out_degrees_h{
        get_vertex_out_degrees(h)
    };
    BOOST_CHECK(expected_out_degrees_h
        == vertex_out_degrees_h
    );
}
```

---

It is expected that  $K_2$  has one out-degree for every vertex, where the two-state Markov chain is expected to have two out-degrees per vertex.

### 3.2 ► Is there an edge between two vertices?

If you have two vertex descriptors, you can check if these are connected by an edge:

---

**Algorithm 38** Check if there exists an edge between two vertices

---

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

template <typename graph>
bool has_edge_between_vertices(
    const typename boost::graph_traits<graph>::
        vertex_descriptor& vd_1,
    const typename boost::graph_traits<graph>::
        vertex_descriptor& vd_2,
    const graph& g
) noexcept
{
    return edge(vd_1, vd_2, g).second;
}
```

---

This code uses the function ‘edge’ (not `boost::edge`: it returns a pair consisting of an edge descriptor and a boolean indicating if it is a valid edge descriptor. The boolean will be true if there exists an edge between the two vertices and false if not.

The demo shows that there is an edge between the two vertices of a  $K_2$  graph, but there are no self-loops (edges that original and end at the same vertex).

---

**Algorithm 39** Demonstration of the ‘has\_edge\_between\_vertices’ function

---

```
#include <boost/test/unit_test.hpp>
#include "create_k2_graph.h"
#include "has_edge_between_vertices.h"

BOOST_AUTO_TEST_CASE(test_has_edge_between_vertices)
{
    const auto g = create_k2_graph();
    const auto vd_1 = *vertices(g).first;
    const auto vd_2 = *(++vertices(g).first);
    BOOST_CHECK( has_edge_between_vertices(vd_1, vd_2, g));
    BOOST_CHECK(!has_edge_between_vertices(vd_1, vd_1, g));
}
```

---

### 3.3 ► Get the edge between two vertices

If you have two vertex descriptors, you can use these to find the edge between them.

---

**Algorithm 40** Get the edge between two vertices

---

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

template <typename graph>
typename boost::graph_traits<graph>::edge_descriptor
get_edge_between_vertices(
    const typename boost::graph_traits<graph>::
        vertex_descriptor& vd_from,
    const typename boost::graph_traits<graph>::
        vertex_descriptor& vd_to,
    const graph& g
)
{
    const auto er = edge(vd_from, vd_to, g);
    if (!er.second)
    {
        std::stringstream msg;
        msg << __func__ << ":_ "
            << "no_edge_between_these_vertices "
            ;
        throw std::invalid_argument(msg.str());
    }
    return er.first;
}
```

---

This code does assume that there is an edge between the two vertices.

The demo shows how to get the edge between two vertices, deleting it, and checking for success.

---

**Algorithm 41** Demonstration of the ‘get\_edge\_between\_vertices’ function

---

```
#include <boost/test/unit_test.hpp>
#include "create_k2_graph.h"
#include "get_edge_between_vertices.h"
#include "has_edge_between_vertices.h"

BOOST_AUTO_TEST_CASE(test_get_edge_between_vertices)
{
    auto g = create_k2_graph();
    const auto vd_1 = *vertices(g).first;
    const auto vd_2 = *(++vertices(g).first);
    BOOST_CHECK(has_edge_between_vertices(vd_1, vd_2, g));
    const auto ed = get_edge_between_vertices(vd_1, vd_2, g);
    boost::remove_edge(ed, g);
    BOOST_CHECK(boost::num_edges(g) == 0);
}
```

---

### 3.4 ►► Create a direct-neighbour subgraph from a vertex descriptor

Suppose you have a vertex of interest its vertex descriptor. Let's say you want to get a subgraph of that vertex and its direct neighbours only. This means that all vertices of that subgraph are adjacent vertices and that the edges go either from focal vertex to its neighbours, or from adjacent vertex to adjacent neighbour.

Here is the ‘create\_direct\_neighbour\_subgraph’ code:

---

**Algorithm 42** Get the direct-neighbour subgraph from a vertex descriptor

---

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

template <typename graph, typename vertex_descriptor>
graph create_direct_neighbour_subgraph(
    const vertex_descriptor& vd,
    const graph& g
)
{
    graph h;

    std::map<vertex_descriptor, vertex_descriptor> m;
    {
        const auto vd_h = boost::add_vertex(h);
        m.insert(std::make_pair(vd, vd_h));
    }
    //Copy vertices
    {
        const auto vdsi = boost::adjacent_vertices(vd, g);
        for (auto i = vdsi.first; i != vdsi.second; ++i)
        {
            if (m.find(*i) == m.end())
            {
                const auto vd_h = boost::add_vertex(h);
                m.insert(std::make_pair(*i, vd_h));
            }
        }
    }
    //Copy edges
    {
        const auto eip = edges(g);
        const auto j = eip.second;
        for (auto i = eip.first; i != j; ++i)
        {
            const auto vd_from = source(*i, g);
            const auto vd_to = target(*i, g);
            if (m.find(vd_from) == std::end(m)) continue;
            if (m.find(vd_to) == std::end(m)) continue;
            boost::add_edge(m[vd_from], m[vd_to], h);
        }
    }
    return h;
}
```

---



This demonstration code shows that the direct-neighbour graph of each vertex of a  $K_2$  graphs is ... a  $K_2$  graph!

---

**Algorithm 43** Demo of the ‘create\_direct\_neighbour\_subgraph’ function

---

```
#include <boost/test/unit_test.hpp>
#include "create_direct_neighbour_subgraph.h"
#include "create_k2_graph.h"

BOOST_AUTO_TEST_CASE(
    test_create_direct_neighbour_subgraph)
{
    const auto g = create_k2_graph();
    const auto vip = vertices(g);
    const auto j = vip.second;
    for (auto i=vip.first; i!=j; ++i) {
        const auto h = create_direct_neighbour_subgraph(
            *i, g
        );
        BOOST_CHECK(boost::num_vertices(h) == 2);
        BOOST_CHECK(boost::num_edges(h) == 1);
    }
}
```

---

Note that this algorithm works on both undirected and directional graphs. If the graph is directional, only the out edges will be copied. To also copy the vertices connected with inward edges, use 3.5

### 3.5 ►► Create a direct-neighbour subgraph from a vertex descriptor including inward edges

Too bad, this algorithm does not work yet.

---

**Algorithm 44** Get the direct-neighbour subgraph from a vertex descriptor

---

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

template <typename graph>
graph create_direct_neighbour_subgraph_including_in_edges
(
    const typename graph::vertex_descriptor& vd, const
    graph& g)
{
    using vertex_descriptor = typename graph::
        vertex_descriptor;
    using edge_descriptor = typename graph::
        edge_descriptor;
    using vpair = std::pair<vertex_descriptor,
        vertex_descriptor>;

    std::vector<vpair> conn_edges;
    std::unordered_map<vertex_descriptor,
        vertex_descriptor> m;

    vertex_descriptor vd_h = 0;
    m.insert(std::make_pair(vd, vd_h++));

    for (const edge_descriptor ed : boost::
        make_iterator_range(edges(g))) {
        const auto vd_from = source(ed, g);
        const auto vd_to = target(ed, g);
        if (vd == vd_from) {
            conn_edges.emplace_back(vd_from, vd_to);
            m.insert(std::make_pair(vd_to, vd_h++));
        }
        if (vd == vd_to) {
            conn_edges.emplace_back(vd_from, vd_to);
            m.insert(std::make_pair(vd_from, vd_h++));
        }
    }

    for (vpair& vp : conn_edges) {
        vp.first = m[vp.first];
        vp.second = m[vp.second];
    }

    return graph(conn_edges.begin(), conn_edges.end(), m,
        size());
}
```

### 3.6 ►► Creating all direct-neighbour subgraphs from a graph without properties

Using the previous function, it is easy to create all direct-neighbour subgraphs from a graph without properties:

---

**Algorithm 45** Create all direct-neighbour subgraphs from a graph without properties

---

```
#include <vector>
#include "create_direct_neighbour_subgraph.h"

template <typename graph>
std::vector<graph> create_all_direct_neighbour_subgraphs(
    const graph& g
) noexcept
{
    using vd = typename graph::vertex_descriptor;

    std::vector<graph> v(boost::num_vertices(g));
    const auto vip = vertices(g);
    std::transform(
        vip.first, vip.second,
        std::begin(v),
        [&g](const vd& d)
        {
            return create_direct_neighbour_subgraph(
                d, g
            );
        }
    );
    return v;
}
```

---

This demonstration code shows that all two direct-neighbour graphs of a  $K_2$  graphs are ...  $K_2$  graphs!

---

**Algorithm 46** Demo of the ‘create\_all\_direct\_neighbour\_subgraphs’ function

---

```
#include <boost/test/unit_test.hpp>
#include "create_all_direct_neighbour_subgraphs.h"
#include "create_k2_graph.h"

BOOST_AUTO_TEST_CASE(
    test_create_all_direct_neighbour_subgraphs)
{
    const auto v
        = create_all_direct_neighbour_subgraphs(
            create_k2_graph());
    BOOST_CHECK(v.size() == 2);
    for (const auto g: v)
    {
        BOOST_CHECK(boost::num_vertices(g) == 2);
        BOOST_CHECK(boost::num_edges(g) == 1);
    }
}
```

---

### 3.7 ► Are two graphs isomorphic?

You may want to check if two graphs are isomorphic. That is: if they have the same shape.

---

**Algorithm 47** Check if two graphs are isomorphic

---

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

template <typename graph1, typename graph2>
bool is_isomorphic(
    const graph1 g,
    const graph2 h
) noexcept
{
    return boost::isomorphism(g, h);
}
```

---

This demonstration code shows that a  $K_3$  graph is not equivalent to a 3-vertices path graph:

---

**Algorithm 48** Demo of the ‘is\_isomorphic’ function

---

```
#include <boost/test/unit_test.hpp>
#include "create_path_graph.h"
#include "create_k3_graph.h"
#include "is_isomorphic.h"

BOOST_AUTO_TEST_CASE(test_is_isomorphic)
{
    const auto g = create_path_graph(3);
    const auto h = create_k3_graph();
    BOOST_CHECK( is_isomorphic(g,g));
    BOOST_CHECK(!is_isomorphic(g,h));
}
```

---

### 3.8 ►► Count the number of connected components in an directed graph

A directed graph may consist out of two components, that are connected within each, but unconnected between them. Take for example, a graph of two isolated edges, with four vertices.

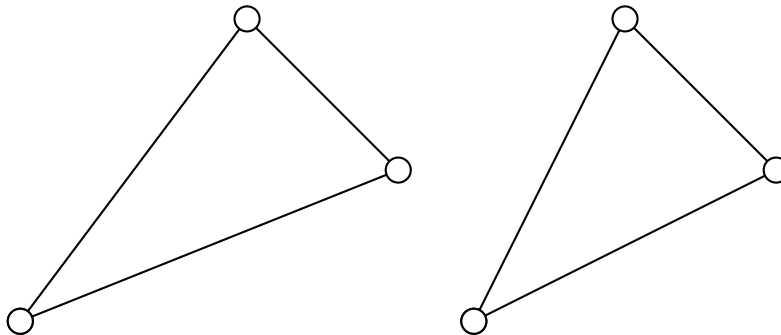


Figure 15: Example of a directed graph with two components

This algorithm counts the number of connected components:

---

**Algorithm 49** Count the number of connected components

---

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

template <typename graph>
int count_directed_graph_connected_components(
    const graph& g
) noexcept
{
    std::vector<int> c(boost::num_vertices(g));
    const int n = boost::strong_components(g,
        boost::make_iterator_property_map(
            std::begin(c),
            get(boost::vertex_index, g)
        )
    );
    return n;
}
```

---

The complexity of this algorithm is  $O(|V| + |E|)$ .

This demonstration code shows that two solitary edges are correctly counted as being two components:

---

**Algorithm 50** Demo of the ‘count\_directed\_graph\_connected\_components’ function

---

```
#include <boost/test/unit_test.hpp>
#include "create_empty_directed_graph.h"
#include "add_edge.h"
#include "count_directed_graph_connected_components.h"

BOOST_AUTO_TEST_CASE(
    test_count_directed_graph_connected_components)
{
    auto g = create_empty_directed_graph();
    BOOST_CHECK(count_directed_graph_connected_components(g) == 0);
    const auto vd_a = boost::add_vertex(g);
    const auto vd_b = boost::add_vertex(g);
    const auto vd_c = boost::add_vertex(g);
    boost::add_edge(vd_a, vd_b, g);
    boost::add_edge(vd_b, vd_c, g);
    boost::add_edge(vd_c, vd_a, g);
    BOOST_CHECK(count_directed_graph_connected_components(g) == 1);
    const auto vd_d = boost::add_vertex(g);
    const auto vd_e = boost::add_vertex(g);
    const auto vd_f = boost::add_vertex(g);
    boost::add_edge(vd_d, vd_e, g);
    boost::add_edge(vd_e, vd_f, g);
    boost::add_edge(vd_f, vd_d, g);
    BOOST_CHECK(count_directed_graph_connected_components(g) == 2);
}
```

---

### 3.9 ►► Count the number of connected components in an undirected graph

An undirected graph may consist out of two components, that are connect within each, but unconnected between them. Take for example, a graph of two isolated edges, with four vertices.

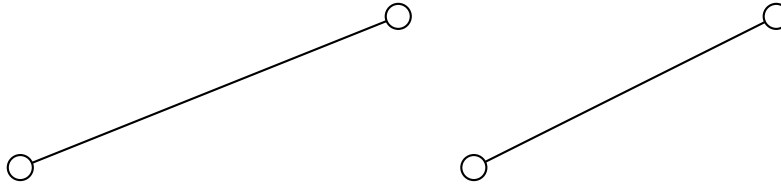


Figure 16: Example of an undirected graph with two components

This algorithm counts the number of connected components:

---

**Algorithm 51** Count the number of connected components

---

```

#include <vector>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/isomorphism.hpp>
#include <boost/graph/connected_components.hpp>

template <typename graph>
int count_undirected_graph_connected_components(
    const graph& g
) noexcept
{
    std::vector<int> c(boost::num_vertices(g));
    return boost::connected_components(g,
        boost::make_iterator_property_map(
            std::begin(c),
            get(boost::vertex_index, g)
        )
    );
}

```

---

The complexity of this algorithm is  $O(|V| + |E|)$ .

This demonstration code shows that two solitary edges are correctly counted as being two components:



---

**Algorithm 52** Demo of the ‘count\_undirected\_graph\_connected\_components’ function

---

```
#include <boost/test/unit_test.hpp>
#include "create_empty_undirected_graph.h"
#include "add_edge.h"
#include "count_undirected_graph_connected_components.h"

BOOST_AUTO_TEST_CASE(
    test_count_undirected_graph_connected_components)
{
    auto g = create_empty_undirected_graph();
    BOOST_CHECK(count_undirected_graph_connected_components
        (g) == 0);
    add_edge(g);
    BOOST_CHECK(count_undirected_graph_connected_components
        (g) == 1);
    add_edge(g);
    BOOST_CHECK(count_undirected_graph_connected_components
        (g) == 2);
}
```

---

### 3.10 ►► Count the number of levels in an undirected graph

Graphs can have a hierarchical structure. From a starting vertex, the number of levels can be counted. A graph of one vertex has zero levels. A graph with one edge has one level. A linear graph of three vertices and two edges has one or two levels, depending on the starting vertex.

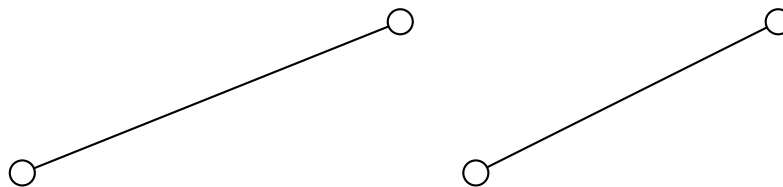


Figure 17: Example of an undirected graph with two components

This algorithm counts the number of levels in an undirected graph, starting at a certain vertex.

It does so, by collecting the neighbours of the traversed vertices. Each sweep, all neighbours of traversed neighbours are added to a set of known vertices. As long as vertices can be added, the algorithm continues. If no vertices can be added, the number of level equals the number of sweeps.

---

**Algorithm 53** Count the number of levels in an undirected graph

---

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

// Collect all neighbours
// If there are no new neighbours, the level is found

template <typename graph>
int count_undirected_graph_levels(
    typename boost::graph_traits<graph>::vertex_descriptor
        vd,
    const graph& g
) noexcept
{
    int level = 0;
    // This does not work:
    // std::set<boost::graph_traits<graph>::
    //     vertex_descriptor> s;
    std::set<int> s;
    s.insert(vd);

    while (1)
    {
        //How many nodes are known now
        const auto sz_before = s.size();

        const auto t = s;

        for (const auto v: t)
        {
            const auto neighbours = boost::adjacent_vertices(v,
                g);
            for (auto n = neighbours.first; n != neighbours.
                second; ++n)
            {
                s.insert(*n);
            }
        }

        //Have new nodes been discovered?
        if (s.size() == sz_before) break;

        //Found new nodes, thus an extra level
        ++level;
    }
    return level;
}
```

This demonstration code shows the number of levels from a certain vertex, while adding edges to form a linear graph. The vertex, when still without edges, has zero levels. After adding one edge, the graph has one level, etc.

---

**Algorithm 54** Demo of the ‘count\_undirected\_graph\_levels’ function

---

```
#include <boost/test/unit_test.hpp>
#include "create_empty_undirected_graph.h"
#include "add_edge.h"
#include "count_undirected_graph_levels.h"

BOOST_AUTO_TEST_CASE(test_count_undirected_graph_levels)
{
    auto g = create_empty_undirected_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 vd_d = boost::add_vertex(g);
    BOOST_CHECK(count_undirected_graph_levels(vd_a, g) ==
        0);
    boost::add_edge(vd_a, vd_b, g);
    BOOST_CHECK(count_undirected_graph_levels(vd_a, g) ==
        1);
    boost::add_edge(vd_b, vd_c, g);
    BOOST_CHECK(count_undirected_graph_levels(vd_a, g) ==
        2);
    boost::add_edge(vd_c, vd_d, g);
    BOOST_CHECK(count_undirected_graph_levels(vd_a, g) ==
        3);
}
```

---

### 3.11 Saving a graph to a .dot file

Graphs 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 (I show .dot file of every non-empty graph created, e.g. chapters 2.14.4 and 2.15.4)

---

**Algorithm 55** Saving a graph to a .dot file

---

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

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 56 shows how to use the ‘`save_graph_to_dot`’ function:

---

**Algorithm 56** Demonstration of the ‘`save_graph_to_dot`’ function

---

```
#include <boost/test/unit_test.hpp>
#include "create_k2_graph.h"
#include "create_markov_chain.h"
#include "save_graph_to_dot.h"

BOOST_AUTO_TEST_CASE(test_save_graph_to_dot)
{
    const auto g = create_k2_graph();
    save_graph_to_dot(g, "create_k2_graph.dot");

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

---

When using the ‘`save_graph_to_dot`’ function (algorithm 55), only the structure of the graph is saved: all other properties like names are not stored. Algorithm ?? shows how to do so.

### 3.12 Loading a 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 57:

---

**Algorithm 57** Loading a directed graph from a .dot file

---

```
#include <fstream>
#include <boost/graph/graphviz.hpp>
#include "create_empty_directed_graph.h"
#include "is_regular_file.h"

boost::adjacency_list<
load_directed_graph_from_dot(
    const std::string& dot_filename
)
{
    if (!is_regular_file(dot_filename))
    {
        std::stringstream msg;
        msg << "_func_" << ":_file_"
            << dot_filename << "':'_not_found"
        ;
        throw std::invalid_argument(msg.str());
    }
    std::ifstream f(dot_filename.c_str());
    auto g = create_empty_directed_graph();
    boost::dynamic_properties dp(
        boost::ignore_other_properties
    );
    boost::read_graphviz(f,g,dp);
    return g;
}
```

---

In this algorithm, first it is checked if the file to load exists, using the ‘is\_regular\_file’ function (algorithm 137), after which an std::ifstream is opened. Then an empty directed graph is created, which saves us writing down the template arguments explicitly. Then, 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 12.5). From this and the empty graph, ‘boost::read\_graphviz’ is called to build up the graph.

Algorithm 58 shows how to use the ‘load\_directed\_graph\_from\_dot’ function:

---

**Algorithm 58** Demonstration of the ‘load\_directed\_graph\_from\_dot’ function

---

```
#include <boost/test/unit_test.hpp>
#include "create_markov_chain.h"
#include "load_directed_graph_from_dot.h"
#include "save_graph_to_dot.h"

BOOST_AUTO_TEST_CASE(test_load_directed_graph_from_dot)
{
    using boost::num_edges;
    using boost::num_vertices;

    const auto g = create_markov_chain();
    const std::string filename{
        "create_markov_chain.dot"
    };
    save_graph_to_dot(g, filename);
    const auto h = load_directed_graph_from_dot(filename);
    BOOST_CHECK(num_edges(g) == num_edges(h));
    BOOST_CHECK(num_vertices(g) == num_vertices(h));
}
```

---

This demonstration shows how the Markov chain is created using the ‘create\_markov\_chain\_graph’ function (algorithm 21), saved and then loaded. The loaded graph is then checked to be a two-state Markov chain.

### 3.13 Loading an undirected graph from a .dot file

Loading an undirected graph from a .dot file is very similar to loading a directed graph from a .dot file, as shown in chapter 3.12. Algorithm 59 show how to do so:

---

**Algorithm 59** Loading an undirected graph from a .dot file

---

```
#include <fstream>
#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
)
{
    if (!is_regular_file(dot_filename))
    {
        std::stringstream msg;
        msg << __func__ << ":_file_"
            << dot_filename << " '_not_found"
        ;
        throw std::invalid_argument(msg.str());
    }
    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;
}
```

---

The only difference with loading a directed graph, is that the initial empty graph is undirected instead. Chapter 3.12 describes the rationale of this function.

Algorithm 60 shows how to use the ‘load\_undirected\_graph\_from\_dot’ function:

---

**Algorithm 60** Demonstration of the ‘load\_undirected\_graph\_from\_dot’ function

---

```
#include <boost/test/unit_test.hpp>
#include "create_k2_graph.h"
#include "load_undirected_graph_from_dot.h"
#include "save_graph_to_dot.h"

BOOST_AUTO_TEST_CASE(test_load_undirected_graph_from_dot)
{
    using boost::num_edges;
    using boost::num_vertices;

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

---

This demonstration shows how the  $K_2$  graph is created using the ‘create\_k2\_graph’ function (algorithm 24), saved and then loaded. The loaded graph is checked to be a  $K_2$  graph.

## 4 Building graphs with bundled vertices

Up until now, the graphs created have had edges and vertices without any properties. In this chapter, graphs will be created, in which the vertices can have a bundled ‘my\_bundled\_vertex’ type<sup>7</sup>. The following graphs will be created:

- An empty directed graph that allows for bundled vertices: see chapter 62
- An empty undirected graph that allows for bundled vertices: see chapter 4.2
- A two-state Markov chain with bundled vertices: see chapter 4.6
- $K_2$  with bundled vertices: see chapter 4.7

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

- Create the vertex class, called ‘my\_bundled\_vertex’: see chapter 4.1
- Adding a ‘my\_bundled\_vertex’: see chapter 4.4

---

<sup>7</sup>I do not intend to be original in naming my data types



- Getting the vertices ‘my\_bundled\_vertex’-es: see chapter 4.5

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

## 4.1 Creating the bundled vertex class

Before creating an empty graph with bundled vertices, that bundled vertex class must be created. In this tutorial, it is called ‘my\_bundled\_vertex’. ‘my\_bundled\_vertex’ is a class that is nonsensical, but it can be replaced by any other class type.

Here I will show the header file of ‘my\_bundled\_vertex’, as the implementation of it is not important:

---

**Algorithm 61** Declaration of my\_bundled\_vertex

---

```
#include <string>
#include <iosfwd>
#include <boost/property_map/dynamic_property_map.hpp>

struct my_bundled_vertex
{
    explicit my_bundled_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;
};

std::ostream& operator<<(std::ostream& os, const
    my_bundled_vertex& e) noexcept;
bool operator==(const my_bundled_vertex& lhs, const
    my_bundled_vertex& rhs) noexcept;
bool operator!=(const my_bundled_vertex& lhs, const
    my_bundled_vertex& rhs) noexcept;
```

---

‘my\_bundled\_vertex’ is a class that has multiple properties:

- It has four public member variables: the double ‘m\_x’ (‘m\_’ stands for member), the double ‘m\_y’, the std::string m\_name and the std::string m\_description. These variables must be public
- It has a default constructor

- It is copyable
- It is comparable for equality (it has `operator==`), which is needed for searching

‘`my_bundled_vertex`’ does not have to have the stream operators defined for file I/O, as this goes via the public member variables.

## 4.2 Create the empty directed graph with bundled vertices

---

**Algorithm 62** Creating an empty directed graph with bundled vertices

---

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

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::directedS,
    my_bundled_vertex
>
create_empty_directed_bundled_vertices_graph() noexcept
{
    return {};
}
```

---

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 directed (due to the `boost::directedS`)
- The vertices have one property: they have a bundled type, that is of data type ‘`my_bundled_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 ‘`my_bundled_vertex`’. This can be read as: “vertices have the bundled property ‘`my_bundled_vertex`’”. Or simply: “vertices have a bundled type called `my_bundled_vertex`”.

### 4.3 Create the empty undirected graph with bundled vertices

---

**Algorithm 63** Creating an empty undirected graph with bundled vertices

---

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

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

---

This code is very similar to the code described in chapter 4.2, except that the directedness (the third template argument) is undirected (due to the `boost::undirectedS`).

### 4.4 Add a bundled vertex

Adding a bundled vertex is very similar to adding a named vertex (chapter ??).

---

**Algorithm 64** Add a bundled vertex

---

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

template <typename graph, typename bundled_vertex>
typename boost::graph_traits<graph>::vertex_descriptor
add_bundled_vertex(const bundled_vertex& v, graph& g)
    noexcept
{
    static_assert(!std::is_const<graph>::value,
        "graph_cannot_be_const");
    return boost::add_vertex(v, g);
}
```

---

When having added a new (abstract) vertex to the graph, the vertex descriptor is used to set the ‘`my_bundled_vertex`’ in the graph.

## 4.5 Getting the bundled vertices' my\_vertexes<sup>8</sup>

When the vertices of a graph have any bundled 'my\_bundled\_vertex', one can extract these as such:

---

**Algorithm 65** Get the bundled vertices' my\_vertexes

---

```
#include <vector>
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/properties.hpp>
#include "my_bundled_vertex.h"

template <typename graph>
std::vector<my_bundled_vertex> get_my_bundled_vertexes(
    const graph& g
) noexcept
{
    using vd = typename graph::vertex_descriptor;

    std::vector<my_bundled_vertex> v(boost::num_vertices(g)
    );
    const auto vip = vertices(g);
    std::transform(vip.first, vip.second, std::begin(v),
        [&g](const vd& d) { return g[d]; }
    );
    return v;
}
```

---

The 'my\_bundled\_vertex' bundled in each vertex is obtained from a vertex descriptor and then put into a std::vector.

The order of the 'my\_bundled\_vertex' objects may be different after saving and loading.

When trying to get the vertices' my\_bundled\_vertex from a graph without these, you will get the error 'formed reference to void' (see chapter 12.1).

## 4.6 Creating a two-state Markov chain with bundled vertices

### 4.6.1 Graph

Figure 18 shows the graph that will be reproduced:

---

<sup>8</sup>the name 'my\_vertexes' is chosen to indicate this function returns a container of my\_vertex

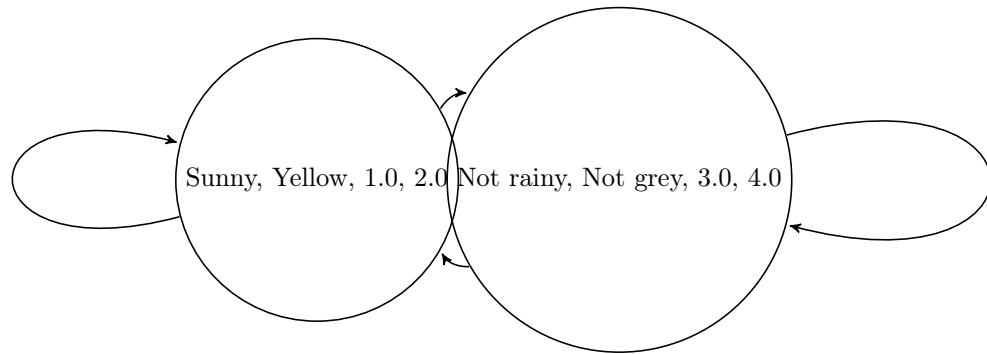


Figure 18: A two-state Markov chain where the vertices have bundled properties and the edges have no properties. The vertices' properties are nonsensical

#### 4.6.2 Function to create such a graph

Here is the code creating a two-state Markov chain with bundled vertices:

---

**Algorithm 66** Creating the two-state Markov chain as depicted in figure 18

---

```
#include "add_bundled_vertex.h"
#include "create_empty_directed_bundled_vertices_graph.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::directedS,
    my_bundled_vertex
>
create_bundled_vertices_markov_chain() noexcept
{
    auto g
        = create_empty_directed_bundled_vertices_graph();
    const my_bundled_vertex a("Sunny",
        "Yellow", 1.0, 2.0
    );
    const my_bundled_vertex b("Not_rainy",
        "Not_grey", 3.0, 4.0
    );
    const auto vd_a = add_bundled_vertex(a, g);
    const auto vd_b = add_bundled_vertex(b, g);
    boost::add_edge(vd_a, vd_a, g);
    boost::add_edge(vd_a, vd_b, g);
    boost::add_edge(vd_b, vd_a, g);
    boost::add_edge(vd_b, vd_b, g);
    return g;
}
```

---

#### 4.6.3 Creating such a graph

Here is the demo:

---

**Algorithm 67** Demo of the ‘create\_bundled\_vertices\_markov\_chain’ function (algorithm 66)

---

```
#include <boost/test/unit_test.hpp>
#include "create_bundled_vertices_markov_chain.h"
#include "get_my_bundled_vertexes.h"
#include "get_my_bundled_vertex.h"

BOOST_AUTO_TEST_CASE(
    test_create_bundled_vertices_markov_chain)
{
    const auto g
        = create_bundled_vertices_markov_chain();
    const std::vector<my_bundled_vertex> expected{
        my_bundled_vertex("Sunny", "Yellow", 1.0, 2.0),
        my_bundled_vertex("Not_rainy", "Not_grey", 3.0, 4.0)
    };
    const auto found = get_my_bundled_vertexes(g);
    BOOST_CHECK(expected == found);
}
```

---

#### 4.6.4 The .dot file produced

---

**Algorithm 68** .dot file created from the ‘create\_bundled\_vertices\_markov\_chain’ function (algorithm 66), converted from graph to .dot file using algorithm 81

---

```
digraph G {
0[label="Sunny",comment="Yellow",width=1,height=2];
1[label="Not$$$SPACE$$$rainy",comment="Not$$$SPACE$$$grey",width=3,height=4];
0->0 ;
0->1 ;
1->0 ;
1->1 ;
}
```

---





#### 4.6.5 The .svg file produced

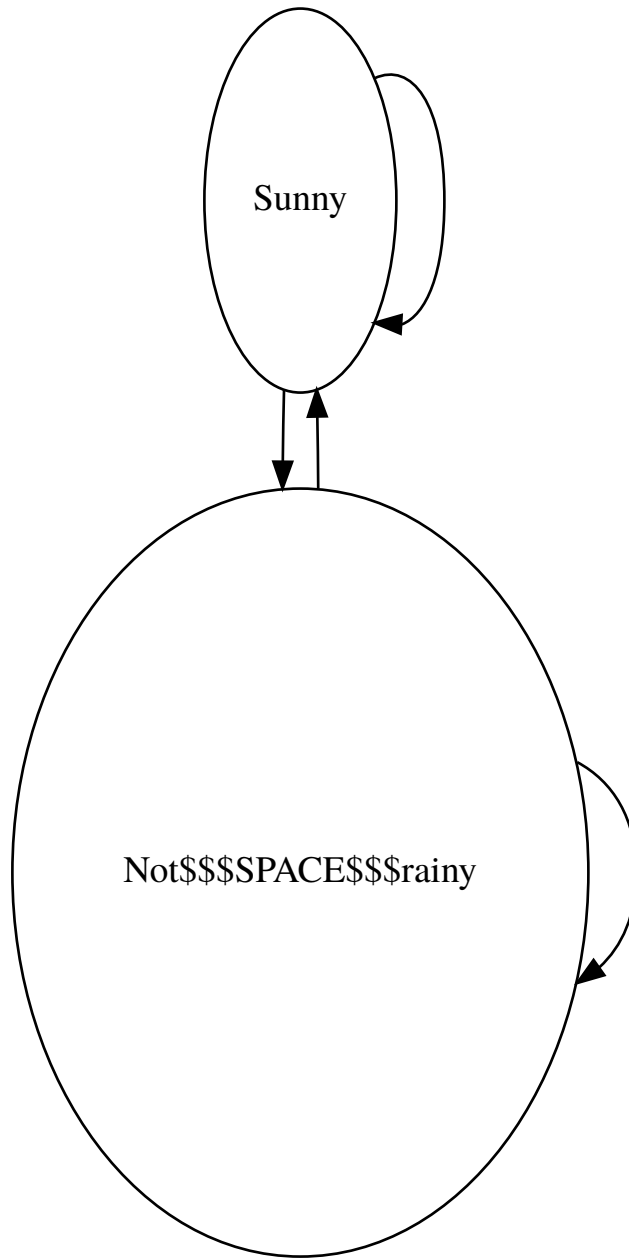


Figure 19: .svg file created from the ‘create\_bundled\_vertices\_markov\_chain’ function (algorithm 66) its .dot file, converted from .dot file to .svg using algorithm 136

## 4.7 Creating $K_2$ with bundled vertices

### 4.7.1 Graph

We reproduce the  $K_2$  with named vertices of chapter ?? , but with our bundled vertices instead, as show in figure 20:

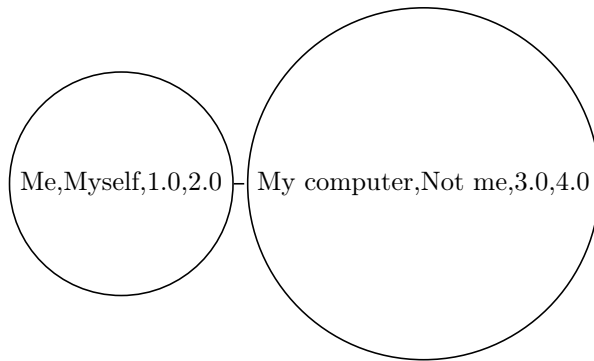


Figure 20:  $K_2$ : a fully connected graph with two bundled vertices

### 4.7.2 Function to create such a graph

---

**Algorithm 69** Creating  $K_2$  as depicted in figure ??

---

```
#include "create_empty_undirected_bundled_vertices_graph.h"
#include "add_bundled_vertex.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    my_bundled_vertex
>
create_bundled_vertices_k2_graph() noexcept
{
    auto g = create_empty_undirected_bundled_vertices_graph
        ();

    const my_bundled_vertex a(
        "Me", "Myself", 1.0, 2.0
    );
    const my_bundled_vertex b(
        "My_computer", "Not_me", 3.0, 4.0
    );
    const auto vd_a = add_bundled_vertex(a, g);
    const auto vd_b = add_bundled_vertex(b, g);
    boost::add_edge(vd_a, vd_b, g);
    return g;
}
```

---

Most of the code is a slight modification of the ‘create\_named\_vertices\_k2\_graph’ function (algorithm ??). In the end, (references to) the my\_bundled\_vertices are obtained and set with two bundled my\_bundled\_vertex objects.

### 4.7.3 Creating such a graph

Demo:

---

**Algorithm 70** Demo of the ‘create\_bundled\_vertices\_k2\_graph’ function (algorithm 69)

---

```
#include <boost/test/unit_test.hpp>
#include "create_bundled_vertices_k2_graph.h"
#include "has_bundled_vertex_with_my_vertex.h"

BOOST_AUTO_TEST_CASE(
    test_create_bundled_vertices_k2_graph)
{
    const auto g = create_bundled_vertices_k2_graph();
    BOOST_CHECK(boost::num_edges(g) == 1);
    BOOST_CHECK(boost::num_vertices(g) == 2);
    BOOST_CHECK(has_bundled_vertex_with_my_vertex(
        my_bundled_vertex("Me", "Myself", 1.0, 2.0), g)
    );
    BOOST_CHECK(has_bundled_vertex_with_my_vertex(
        my_bundled_vertex("My_computer", "Not_me", 3.0, 4.0), g)
    );
}
```

---

#### 4.7.4 The .dot file produced

---

**Algorithm 71** .dot file created from the ‘create\_bundled\_vertices\_k2\_graph’ function (algorithm 69), converted from graph to .dot file using algorithm 55

---

```
graph G {
0[label="Me",comment="Myself",width=1,height=2];
1[label="My$$$SPACE$$$computer",comment="Not$$$SPACE$$$me",width=3,height=4];
0--1 ;
}
```

---



#### 4.7.5 The .svg file produced

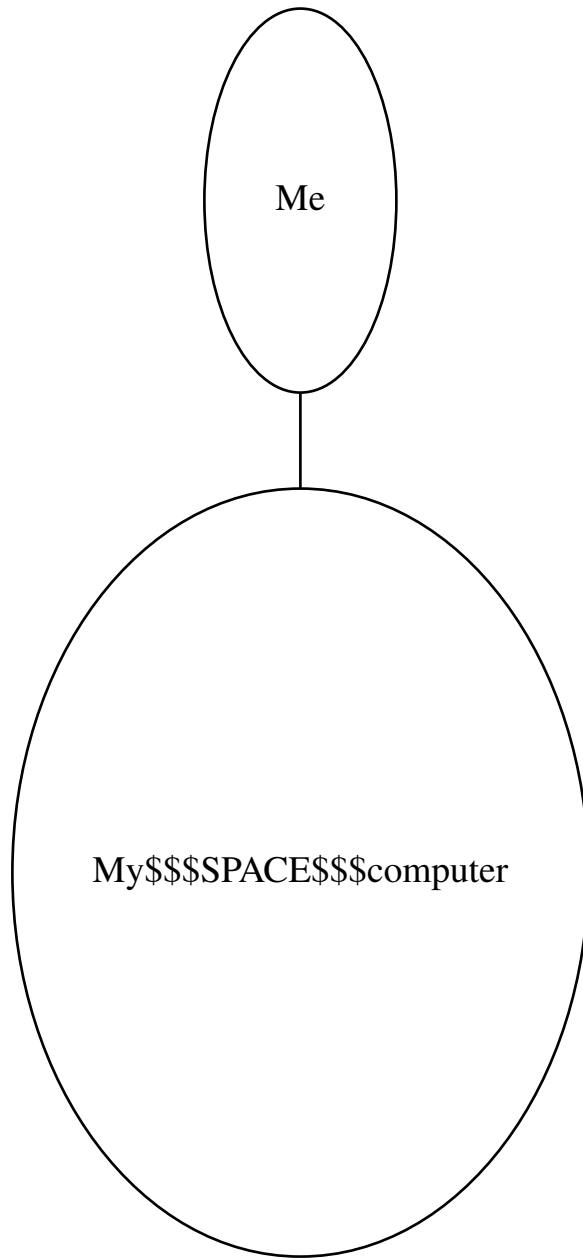


Figure 21: .svg file created from the 'create\_bundled\_vertices\_k2\_graph' function (algorithm 69) its .dot file, converted from .dot file to .svg using algorithm 136

## 5 Working on graphs with bundled vertices

When using graphs with bundled vertices, their state gives a way to find a vertex and working with it. This chapter shows some basic operations on graphs with bundled vertices.

- Check if there exists a vertex with a certain ‘my\_bundled\_vertex’: chapter 5.1
- Find a vertex with a certain ‘my\_bundled\_vertex’: chapter 5.2
- Get a vertex its ‘my\_bundled\_vertex’ from its vertex descriptor: chapter 5.3
- Set a vertex its ‘my\_bundled\_vertex’ using its vertex descriptor: chapter 5.4
- Setting all vertices their ‘my\_bundled\_vertex’-es: chapter 5.5
- Storing an directed/undirected graph with bundled vertices as a .dot file: chapter 5.6
- Loading a directed graph with bundled vertices from a .dot file: chapter 5.7
- Loading an undirected directed graph with bundled vertices from a .dot file: chapter 5.8

### 5.1 Has a bundled vertex with a my\_bundled\_vertex

Before modifying our vertices, let’s first determine if we can find a vertex by its bundled type (‘my\_bundled\_vertex’) in a graph. After obtain the vertex iterators, we can dereference each these to obtain the vertex descriptors and then compare each vertex its ‘my\_bundled\_vertex’ with the one desired.

---

**Algorithm 72** Find if there is vertex with a certain `my_bundled_vertex`

---

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

template <typename graph>
bool has_bundled_vertex_with_my_vertex(
    const my_bundled_vertex& v,
    const graph& g
) noexcept
{
    using vd = typename graph::vertex_descriptor;

    const auto vip = vertices(g);
    return std::find_if(vip.first, vip.second,
        [&v, &g](const vd& d)
        {
            return g[d] == v;
        }
    ) != vip.second;
}
```

---

This function can be demonstrated as in algorithm 73, where a certain `my_bundled_vertex` cannot be found in an empty graph. After adding the desired `my_bundled_vertex`, it is found.



---

**Algorithm 73** Demonstration of the ‘has\_bundled\_vertex\_with\_my\_vertex’ function

---

```
#include <boost/test/unit_test.hpp>
#include "add_bundled_vertex.h"
#include "create_empty_undirected_bundled_vertices_graph.h"
#include "has_bundled_vertex_with_my_vertex.h"
#include "my_bundled_vertex.h"

BOOST_AUTO_TEST_CASE(
    test_has_bundled_vertex_with_my_vertex)
{
    auto g = create_empty_undirected_bundled_vertices_graph();
    BOOST_CHECK(!has_bundled_vertex_with_my_vertex(
        my_bundled_vertex("Felix"), g));
    add_bundled_vertex(my_bundled_vertex("Felix"), g);
    BOOST_CHECK(has_bundled_vertex_with_my_vertex(
        my_bundled_vertex("Felix"), g));
}
```

---

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

## 5.2 Find a bundled vertex with a certain my\_bundled\_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 74 shows how to obtain a vertex descriptor to the first vertex found with a specific ‘my\_bundled\_vertex’ value.

---

**Algorithm 74** Find the first vertex with a certain `my_bundled_vertex`

---

```
#include <cassert>
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/properties.hpp>
#include "has_bundled_vertex_with_my_vertex.h"
#include "my_bundled_vertex.h"

template <typename graph, typename bundled_vertex_t>
typename boost::graph_traits<graph>::vertex_descriptor
find_first_bundled_vertex_with_my_vertex(
    const bundled_vertex_t& v,
    const graph& g
)
{
    using vd = typename graph::vertex_descriptor;
    const auto vip = vertices(g);
    const auto i = std::find_if(
        vip.first, vip.second,
        [&v, &g](const vd d) { return g[d] == v; }
    );
    if (i == vip.second)
    {
        std::stringstream msg;
        msg << __func__ << ":_ "
            << "could_not_find_my_bundled_vertex_"
            << v << "' "
            ;
        throw std::invalid_argument(msg.str());
    }
    return *i;
}
```

---

With the vertex descriptor obtained, one can read and modify the vertex and the edges surrounding it. Algorithm 75 shows some examples of how to do so.

---

**Algorithm 75** Demonstration of the ‘find\_first\_bundled\_vertex\_with\_my\_vertex’ function

---

```
#include <boost/test/unit_test.hpp>
#include "create_bundled_vertices_k2_graph.h"
#include "find_first_bundled_vertex_with_my_vertex.h"

BOOST_AUTO_TEST_CASE(
    test_find_first_bundled_vertex_with_my_vertex)
{
    const auto g = create_bundled_vertices_k2_graph();
    const auto vd =
        find_first_bundled_vertex_with_my_vertex(
            my_bundled_vertex("Me", "Myself", 1.0, 2.0),
            g
        );
    BOOST_CHECK(out_degree(vd, g) == 1);
    BOOST_CHECK(in_degree(vd, g) == 1);
}
```

---

### 5.3 Get a bundled vertex its ‘my\_bundled\_vertex’

To obtain the ‘my\_bundled\_vertex’ from a vertex descriptor is simple:

---

**Algorithm 76** Get a bundled vertex its my\_vertex from its vertex descriptor

---

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

template <typename graph>
my_bundled_vertex get_my_bundled_vertex(
    const typename boost::graph_traits<graph>::
        vertex_descriptor& vd,
    const graph& g
) noexcept
{
    return g[vd];
}
```

---

One can just use the graph as a property map and let it be looked-up.

To use ‘get\_bundled\_vertex\_my\_vertex’, one first needs to obtain a vertex descriptor. Algorithm 77 shows a simple example.

---

**Algorithm 77** Demonstration if the ‘get\_bundled\_vertex\_my\_vertex’ function

---

```
#include <boost/test/unit_test.hpp>
#include "add_bundled_vertex.h"
#include "create_empty_undirected_bundled_vertices_graph.h"
#include "find_first_bundled_vertex_with_my_vertex.h"
#include "get_my_bundled_vertex.h"

BOOST_AUTO_TEST_CASE(test_get_my_bundled_vertex)
{
    auto g
        = create_empty_undirected_bundled_vertices_graph();
    const my_bundled_vertex v{"Dex"};
    add_bundled_vertex(v, g);
    const auto vd
        = find_first_bundled_vertex_with_my_vertex(v, g);
    BOOST_CHECK(get_my_bundled_vertex(vd, g) == v);
}
```

---

## 5.4 Set a bundled vertex its my\_vertex

If you know how to get the ‘my\_bundled\_vertex’ from a vertex descriptor, setting it is just as easy, as shown in algorithm 78.

---

**Algorithm 78** Set a bundled vertex its my\_vertex from its vertex descriptor

---

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

template <typename graph>
void set_my_bundled_vertex(
    const my_bundled_vertex& v,
    const typename boost::graph_traits<graph>::
        vertex_descriptor& vd,
    graph& g
) noexcept
{
    static_assert(!std::is_const<graph>::value, "graph_
        cannot_be_const");
    g[vd] = v;
}
```

---

To use ‘set\_bundled\_vertex\_my\_vertex’, one first needs to obtain a vertex descriptor. Algorithm 79 shows a simple example.

---

**Algorithm 79** Demonstration if the ‘set\_bundled\_vertex\_my\_vertex’ function

---

```
#include <boost/test/unit_test.hpp>
#include "add_bundled_vertex.h"
#include "create_empty_undirected_bundled_vertices_graph.h"
#include "find_first_bundled_vertex_with_my_vertex.h"
#include "get_my_bundled_vertex.h"
#include "set_my_bundled_vertex.h"

BOOST_AUTO_TEST_CASE(test_set_my_bundled_vertex)
{
    auto g = create_empty_undirected_bundled_vertices_graph();
    const my_bundled_vertex old_name{"Dex"};
    add_bundled_vertex(old_name, g);
    const auto vd =
        find_first_bundled_vertex_with_my_vertex(old_name, g);
    BOOST_CHECK(get_my_bundled_vertex(vd, g) == old_name);
    const my_bundled_vertex new_name{"Diggy"};
    set_my_bundled_vertex(new_name, vd, g);
    BOOST_CHECK(get_my_bundled_vertex(vd, g) == new_name);
}
```

---

## 5.5 Setting all bundled vertices’ my\_vertex objects

When the vertices of a graph are ‘my\_bundled\_vertex’ objects, one can set these as such:

---

**Algorithm 80** Setting the bundled vertices' 'my\_bundled\_vertex'-es

---

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

template <typename graph>
void set_my_bundled_vertexes(
    graph& g,
    const std::vector<my_bundled_vertex>& my_vertexes
) noexcept
{
    static_assert(!std::is_const<graph>::value,
        "graph_cannot_be_const");

    auto my_vertexes_begin = std::begin(my_vertexes);
    //const auto my_vertexes_end = std::end(my_vertexes);
    const auto vip = vertices(g);
    const auto j = vip.second;
    for (
        auto i = vip.first;
        i!=j; ++i,
        ++my_vertexes_begin
    ) {
        //assert(my_vertexes_begin != my_vertexes_end);
        g[*i] = *my_vertexes_begin;
    }
}
```

---

## 5.6 Storing a graph with bundled vertices as a .dot

If you used the 'create\_bundled\_vertices\_k2\_graph' function (algorithm 69) to produce a  $K_2$  graph with vertices associated with 'my\_bundled\_vertex' objects, you can store these with algorithm 81:

---

**Algorithm 81** Storing a graph with bundled vertices as a .dot file

---

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

template <typename graph>
void save_bundled_vertices_graph_to_dot(
    const graph& g,
    const std::string& filename
)
{
    std::ofstream f(filename);
    boost::write_graphviz(f, g,
        make_bundled_vertices_writer(g)
    );
}
```

---

This code looks small, because we call the ‘make\_bundled\_vertices\_writer’ function, which is shown in algorithm 82:

---

**Algorithm 82** The ‘make\_bundled\_vertices\_writer’ function

---

```
template <typename graph>
inline bundled_vertices_writer<graph>
make_bundled_vertices_writer(
    const graph& g
)
{
    return bundled_vertices_writer<
        graph
    >(g);
}
```

---

Also this function is forwarding the real work to the ‘bundled\_vertices\_writer’, shown in algorithm 83:

---

**Algorithm 83** The ‘bundled\_vertices\_writer’ function

---

```
#include <ostream>
#include "graphviz_encode.h"
#include "is_graphviz_friendly.h"

template <
    typename graph
>
class bundled_vertices_writer {
public:
    bundled_vertices_writer(
        graph g
    ) : m_g{g}
    {

    }

    template <class vertex_descriptor>
    void operator()(
        std::ostream& out,
        const vertex_descriptor& vd
    ) const noexcept {
        out
            << "[label=\"\"
            << graphviz_encode(
                m_g[vd].m_name
            )
            << "\",comment=\"\"
            << graphviz_encode(
                m_g[vd].m_description
            )
            << "\",width=\"
            << m_g[vd].m_x
            << "\",height=\"
            << m_g[vd].m_y
            << "]"
        ;
    }
private:
    graph m_g;
};
```

---

Here, some interesting things are happening: the writer needs the bundled property maps to work with and thus copies the whole graph to its internals. I have chosen to map the ‘my\_bundled\_vertex’ member variables to Graphviz



attributes (see chapter 13.2 for most Graphviz attributes) as shown in table 1:

my_bundled_vertex variable	C++ data type	Graphviz data type	Graphviz attribute
m_name	std::string	string	label
m_description	std::string	string	comment
m_x	double	double	width
m_y	double	double	height

Table 1: Mapping of my\_bundled\_vertex member variable and Graphviz attributes

Important in this mapping is that the C++ and the Graphviz data types match. I also chose attributes that matched as closely as possible.

The writer also encodes the std::string of the name and description to a Graphviz-friendly format. When loading the .dot file again, this will have to be undone again.

## 5.7 Loading a directed graph with bundled vertices from a .dot

When loading a graph from file, one needs to specify a type of graph. In this example, an directed graph with bundled vertices is loaded, as shown in algorithm 84:

---

**Algorithm 84** Loading a directed graph with bundled vertices from a .dot file

---

```
#include <fstream>
#include <boost/graph/graphviz.hpp>
#include "create_empty_directed_bundled_vertices_graph.h"
#include "graphviz_decode.h"
#include "is_regular_file.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::directedS,
    my_bundled_vertex
>
load_directed_bundled_vertices_graph_from_dot(
    const std::string& dot_filename
)
{
    if (!is_regular_file(dot_filename))
    {
        std::stringstream msg;
        msg << __func__ << ":_file_"
            << dot_filename << "':_not_found"
        ;
        throw std::invalid_argument(msg.str());
    }
    std::ifstream f(dot_filename.c_str());
    auto g = create_empty_directed_bundled_vertices_graph()
        ;

    boost::dynamic_properties dp(boost::
        ignore_other_properties);
    dp.property("label", get(&my_bundled_vertex::m_name, g))
        ;
    dp.property("comment", get(&my_bundled_vertex::
        m_description, g));
    dp.property("width", get(&my_bundled_vertex::m_x, g));
    dp.property("height", get(&my_bundled_vertex::m_y, g));
    boost::read_graphviz(f, g, dp);

    //Decode vertices
    const auto vip = vertices(g);
    const auto j = vip.second;
    for (auto i = vip.first; i!=j; ++i)
    {
        g[*i].m_name = graphviz_decode(g[*i].m_name);
        g[*i].m_description = graphviz_decode(g[*i].
            m_description);
    }
    90

    return g;
}
```

---

In this algorithm, first it is checked if the file to load exists. Then an empty directed graph is created, to save typing the typename explicitly.

Then a `boost::dynamic_properties` is created with its default constructor, after which we set it to follow the same mapping as in the previous chapter. From this and the empty graph, `'boost::read_graphviz'` is called to build up the graph.

At the moment the graph is created, all `'my_bundled_vertex'` their names and description are in a Graphviz-friendly format. By obtaining all vertex iterators and vertex descriptors, the encoding is made undone.

Algorithm 85 shows how to use the `'load_directed_bundled_vertices_graph_from_dot'` function:

---

**Algorithm 85** Demonstration of the `'load_directed_bundled_vertices_graph_from_dot'` function

---

```
#include <boost/test/unit_test.hpp>
#include "create_bundled_vertices_markov_chain.h"
#include "load_directed_bundled_vertices_graph_from_dot.h"
"

#include "save_bundled_vertices_graph_to_dot.h"
#include "get_my_bundled_vertexes.h"

BOOST_AUTO_TEST_CASE(
    test_load_directed_bundled_vertices_graph_from_dot)
{
    using boost::num_edges;
    using boost::num_vertices;

    const auto g
        = create_bundled_vertices_markov_chain();
    const std::string filename{
        "create_bundled_vertices_markov_chain.dot"
    };
    save_bundled_vertices_graph_to_dot(g, filename);
    const auto h
        = load_directed_bundled_vertices_graph_from_dot(
            filename);
    BOOST_CHECK(num_edges(g) == num_edges(h));
    BOOST_CHECK(num_vertices(g) == num_vertices(h));
    BOOST_CHECK(get_my_bundled_vertexes(g) ==
        get_my_bundled_vertexes(h));
}
```

---

This demonstration shows how the Markov chain is created using the `'create_bundled_vertices_markov_chain'` function (algorithm 66), saved and then

loaded. The loaded graph is checked to be the same as the original.

### **5.8 Loading an undirected graph with bundled vertices from a .dot**

When loading a graph from file, one needs to specify a type of graph. In this example, an undirected graph with bundled vertices is loaded, as shown in algorithm 86:

---

**Algorithm 86** Loading an undirected graph with bundled vertices from a .dot file

---

```

#include <fstream>
#include <boost/graph/graphviz.hpp>
#include "create_empty_undirected_bundled_vertices_graph.
    h"
#include "graphviz_decode.h"
#include "is_regular_file.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    my_bundled_vertex
>
load_undirected_bundled_vertices_graph_from_dot(
    const std::string& dot_filename
)
{
    if (!is_regular_file(dot_filename))
    {
        std::stringstream msg;
        msg << __func__ << ":_file_"
            << dot_filename << " '_not_found"
            ;
        throw std::invalid_argument(msg.str());
    }
    std::ifstream f(dot_filename.c_str());
    auto g = create_empty_undirected_bundled_vertices_graph
        ();

    boost::dynamic_properties dp(boost::
        ignore_other_properties);
    dp.property("label", get(&my_bundled_vertex::m_name, g))
        ;
    dp.property("comment", get(&my_bundled_vertex::
        m_description, g));
    dp.property("width", get(&my_bundled_vertex::m_x, g));
    dp.property("height", get(&my_bundled_vertex::m_y, g));
    boost::read_graphviz(f, g, dp);

    //Decode vertices
    const auto vip = vertices(g);
    const auto j = vip.second;
    for (auto i = vip.first; i!=j; ++i)
    {
        g[*i].m_name = graphviz_decode(g[*i].m_name);
        g[*i].m_description = graphviz_decode(g[*i].
            m_description);
    }

    return g;
}

```

---

The only difference with loading a directed graph, is that the initial empty graph is undirected instead. Chapter 5.7 describes the rationale of this function.

Algorithm 87 shows how to use the ‘load\_undirected\_bundled\_vertices\_graph\_from\_dot’ function:

---

**Algorithm 87** Demonstration of the ‘load\_undirected\_bundled\_vertices\_graph\_from\_dot’ function

---

```
#include <boost/test/unit_test.hpp>
#include "create_bundled_vertices_k2_graph.h"
#include "load_undirected_bundled_vertices_graph_from_dot
.h"
#include "save_bundled_vertices_graph_to_dot.h"
#include "get_my_bundled_vertexes.h"

BOOST_AUTO_TEST_CASE(
    test_load_undirected_bundled_vertices_graph_from_dot)
{
    using boost::num_edges;
    using boost::num_vertices;

    const auto g
        = create_bundled_vertices_k2_graph();
    const std::string filename{
        "create_bundled_vertices_k2_graph.dot"
    };
    save_bundled_vertices_graph_to_dot(g, filename);
    const auto h
        = load_undirected_bundled_vertices_graph_from_dot(
            filename);
    BOOST_CHECK(get_my_bundled_vertexes(g)
        == get_my_bundled_vertexes(h)
    );
}
```

---

This demonstration shows how  $K_2$  with bundled vertices is created using the ‘create\_bundled\_vertices\_k2\_graph’ function (algorithm 69), saved and then loaded. The loaded graph is checked to be the same as the original.

## 6 Building graphs with bundled edges and vertices

Up until now, the graphs created have had only bundled vertices. In this chapter, graphs will be created, in which both the edges and vertices have a bundled

‘my\_bundled\_edge’ and ‘my\_bundled\_edge’ type<sup>9</sup>.

- An empty directed graph that allows for bundled edges and vertices: see chapter 6.2
- An empty undirected graph that allows for bundled edges and vertices: see chapter 6.3
- A two-state Markov chain with bundled edges and vertices: see chapter 6.6
- $K_3$  with bundled edges and vertices: see chapter 6.7

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

- Creating the ‘my\_bundled\_edge’ class: see chapter 6.1
- Adding a bundled ‘my\_bundled\_edge’: see chapter 6.4

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

## 6.1 Creating the bundled edge class

In this example, I create a ‘my\_bundled\_edge’ class. Here I will show the header file of it, as the implementation of it is not important yet.

---

<sup>9</sup>I do not intend to be original in naming my data types

---

**Algorithm 88** Declaration of `my_bundled_edge`

---

```
#include <string>
#include <iosfwd>

class my_bundled_edge
{
public:
    explicit my_bundled_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;
};

std::ostream& operator<<(std::ostream& os, const
    my_bundled_edge& e) noexcept;
bool operator==(const my_bundled_edge& lhs, const
    my_bundled_edge& rhs) noexcept;
bool operator!=(const my_bundled_edge& lhs, const
    my_bundled_edge& rhs) noexcept;
```

---

`my_bundled_edge` is a class that has multiple properties: two doubles ‘`m_width`’ (‘`m_`’ stands for member) and ‘`m_height`’, and two `std::string`s `m_name` and `m_description`. ‘`my_bundled_edge`’ is copyable, but cannot trivially be converted to a ‘`std::string`.’ ‘`my_bundled_edge`’ is comparable for equality (that is, `operator==` is defined).

‘`my_bundled_edge`’ does not have to have the stream operators defined for file I/O, as this goes via the public member variables.



## 6.2 Create an empty directed graph with bundled edges and vertices

---

**Algorithm 89** Creating an empty directed graph with bundled edges and vertices

---

```
#include <boost/graph/adjacency_list.hpp>
#include "my_bundled_edge.h"
#include "my_bundled_vertex.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::directedS,
    my_bundled_vertex,
    my_bundled_edge
>
create_empty_directed_bundled_edges_and_vertices_graph()
    noexcept
{
    return {};
}
```

---

This code is very similar to the code described in chapter ??, except that there is a new, fifth template argument:

```
boost::property<boost::edge_bundled_type_t, my_edge>
```

This can be read as: “edges have the property ‘boost::edge\_bundled\_type\_t’, which is of data type ‘my\_bundled\_edge’”. Or simply: “edges have a bundled type called my\_bundled\_edge”.

Demo:

---

**Algorithm 90** Demonstration of the ‘create\_empty\_directed\_bundled\_edges\_and\_vertices\_graph’ function

---

```
#include <boost/test/unit_test.hpp>
#include "
    create_empty_directed_bundled_edges_and_vertices_graph
    .h"

BOOST_AUTO_TEST_CASE(
    test_create_empty_directed_bundled_edges_and_vertices_graph
)
{
    const auto g =
        create_empty_directed_bundled_edges_and_vertices_graph
        ();
    BOOST_CHECK(boost::num_edges(g) == 0);
    BOOST_CHECK(boost::num_vertices(g) == 0);
}
```

---

### 6.3 Create an empty undirected graph with bundled edges and vertices

---

**Algorithm 91** Creating an empty undirected graph with bundled edges and vertices

---

```
#include <boost/graph/adjacency_list.hpp>
#include "my_bundled_edge.h"
#include "my_bundled_vertex.h"

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

---

This code is very similar to the code described in chapter 6.2, except that the directness (the third template argument) is undirected (due to the `boost::undirectedS`).

Demo:

---

**Algorithm 92** Demonstration of the ‘create\_empty\_undirected\_bundled\_edges\_and\_vertices\_graph’ function

---

```
#include <boost/test/unit_test.hpp>
#include "
    create_empty_undirected_bundled_edges_and_vertices_graph
    .h"

BOOST_AUTO_TEST_CASE(
    test_create_empty_undirected_bundled_edges_and_vertices_graph
)
{
    const auto g
        =
            create_empty_undirected_bundled_edges_and_vertices_graph
            ();
    BOOST_CHECK(boost::num_edges(g) == 0);
    BOOST_CHECK(boost::num_vertices(g) == 0);
}
```

---

## 6.4 Add a bundled edge

Adding a bundled edge is very similar to adding a named edge (chapter ??).

---

**Algorithm 93** Add a bundled edge

---

```
#include <cassert>
#include <sstream>
#include <stdexcept>
#include <boost/graph/adjacency_list.hpp>
#include "my_bundled_edge.h"
#include "has_edge_between_vertices.h"

template <typename graph, typename bundled_edge>
typename boost::graph_traits<graph>::edge_descriptor
add_bundled_edge(
    const typename boost::graph_traits<graph>::
        vertex_descriptor& vd_from,
    const typename boost::graph_traits<graph>::
        vertex_descriptor& vd_to,
    const bundled_edge& edge,
    graph& g
)
{
    static_assert(!std::is_const<graph>::value, "graph_
        cannot_be_const");
    if (has_edge_between_vertices(vd_from, vd_to, g))
    {
        std::stringstream msg;
        msg << __func__ << ":_already_an_edge_there";
        throw std::invalid_argument(msg.str());
    }
    const auto aer = boost::add_edge(vd_from, vd_to, g);
    assert(aer.second);
    g[aer.first] = edge;
    return aer.first;
}
```

---

When having added a new (abstract) edge to the graph, the edge descriptor is used to set the `my_edge` in the graph.

Here is the demo:

---

**Algorithm 94** Demo of ‘add\_bundled\_edge’

---

```
#include <boost/test/unit_test.hpp>
#include "add_bundled_edge.h"
#include "add_bundled_vertex.h"
#include "
    create_empty_directed_bundled_edges_and_vertices_graph
    .h"

BOOST_AUTO_TEST_CASE(test_add_bundled_edge)
{
    auto g =
        create_empty_directed_bundled_edges_and_vertices_graph
        ();
    const auto vd_from = add_bundled_vertex(
        my_bundled_vertex("From"), g);
    const auto vd_to = add_bundled_vertex(my_bundled_vertex
        ("To"), g);
    add_bundled_edge(vd_from, vd_to, my_bundled_edge("X"),
        g);
    BOOST_CHECK(boost::num_vertices(g) == 2);
    BOOST_CHECK(boost::num_edges(g) == 1);
}
```

---

## 6.5 Getting the bundled edges my\_edges

When the edges of a graph are ‘my\_bundled\_edge’ objects, one can extract these all as such:

---

**Algorithm 95** Get the edges' my\_bundled\_edges

---

```
#include <vector>
#include <boost/graph/adjacency_list.hpp>
#include "my_bundled_edge.h"

template <typename graph>
std::vector<my_bundled_edge> get_my_bundled_edges(
    const graph& g
) noexcept
{
    using ed = typename boost::graph_traits<graph>::
        edge_descriptor;
    std::vector<my_bundled_edge> v(boost::num_edges(g));
    const auto eip = edges(g);
    std::transform(eip.first, eip.second, std::begin(v),
        [&g](const ed e) { return g[e]; }
    );
    return v;
}
```

---

The 'my\_bundled\_edge' object associated with the edges are obtained from the graph its property\_map and then put into a std::vector.

Note: the order of the my\_bundled\_edge objects may be different after saving and loading.

When trying to get the edges' my\_bundled\_edge objects from a graph without bundled edges objects associated, you will get the error 'formed reference to void' (see chapter 12.1).

## 6.6 Creating a Markov-chain with bundled edges and vertices

### 6.6.1 Graph

Figure 22 shows the graph that will be reproduced:

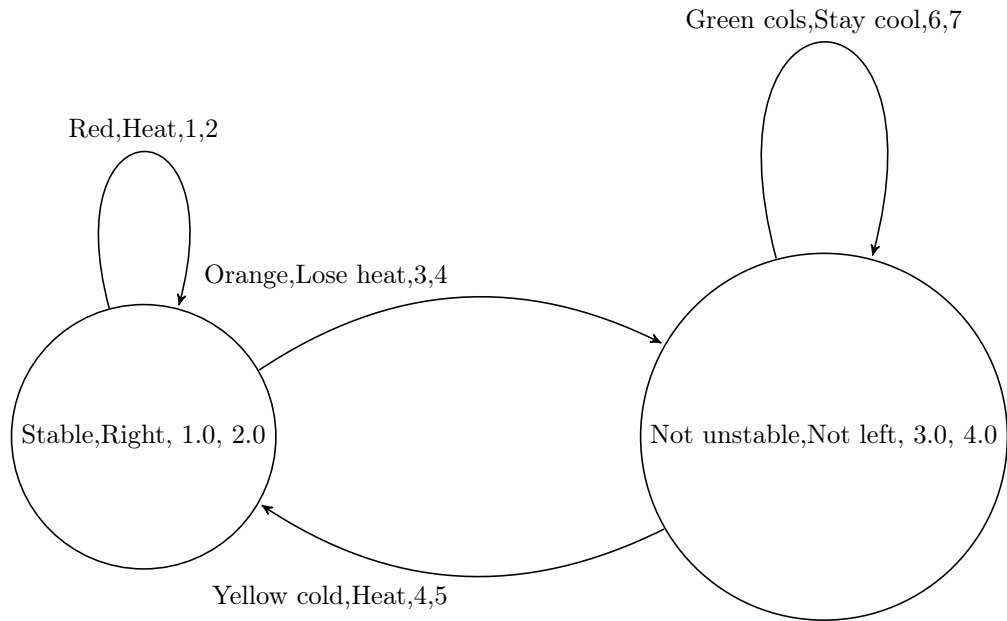


Figure 22: A two-state Markov chain where the edges and vertices have bundled properties. The edges' and vertices' properties are nonsensical

### 6.6.2 Function to create such a graph

Here is the code creating a two-state Markov chain with bundled edges and vertices:

---

**Algorithm 96** Creating the two-state Markov chain as depicted in figure 22

---

```

#include <cassert>
#include "
    create_empty_directed_bundled_edges_and_vertices_graph
    .h"
#include "add_bundled_edge.h"
#include "add_bundled_vertex.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::directedS,
    my_bundled_vertex,
    my_bundled_edge
>
create_bundled_edges_and_vertices_markov_chain()
{
    auto g
        =
            create_empty_directed_bundled_edges_and_vertices_graph
            ();
    const auto va = my_bundled_vertex("Stable","Right"
        ,1.0,2.0);
    const auto vb = my_bundled_vertex("Not_unstable","Not_
        left",3.0,4.0);
    const auto vd_a = add_bundled_vertex(va, g);
    const auto vd_b = add_bundled_vertex(vb, g);
    const auto e_aa = my_bundled_edge("Red","Heat",1.0,2.0)
        ;
    const auto e_ab = my_bundled_edge("Orange","Lose_heat"
        ,3.0,4.0);
    const auto e_ba = my_bundled_edge("Yellow_cold","Heat"
        ,5.0,6.0);
    const auto e_bb = my_bundled_edge("Green_cold","Stay_
        cool",7.0,8.0);
    add_bundled_edge(vd_a, vd_a, e_aa, g);
    add_bundled_edge(vd_a, vd_b, e_ab, g);
    add_bundled_edge(vd_b, vd_a, e_ba, g);
    add_bundled_edge(vd_b, vd_b, e_bb, g);
    return g;
}

```

---



### 6.6.3 Creating such a graph

Here is the demo:

---

**Algorithm 97** Demo of the ‘create\_bundled\_edges\_and\_vertices\_markov\_chain’ function (algorithm 96)

---

```
#include <boost/test/unit_test.hpp>
#include "create_bundled_edges_and_vertices_markov_chain.h"
#include "get_my_bundled_edges.h"
#include "my_bundled_vertex.h"

BOOST_AUTO_TEST_CASE(
    test_create_bundled_edges_and_vertices_markov_chain)
{
    const auto g =
        create_bundled_edges_and_vertices_markov_chain();
    const std::vector<my_bundled_edge> edge_my_edges{
        get_my_bundled_edges(g)
    };
    const std::vector<my_bundled_edge> expected_my_edges{
        my_bundled_edge("Red", "Heat", 1.0, 2.0),
        my_bundled_edge("Orange", "Lose_heat", 3.0, 4.0),
        my_bundled_edge("Yellow_cold", "Heat", 5.0, 6.0),
        my_bundled_edge("Green_cold", "Stay_cool", 7.0, 8.0)
    };
    BOOST_CHECK(edge_my_edges == expected_my_edges);
}
```

---

### 6.6.4 The .dot file produced

---

**Algorithm 98** .dot file created from the ‘create\_bundled\_edges\_and\_vertices\_markov\_chain’ function (algorithm 96), converted from graph to .dot file using algorithm 55

---

```
digraph G {
0[label="Stable",comment="Right",width=1,height=2];
1[label="Not$$$$SPACE$$$unstable",comment="Not$$$$SPACE$$$left",width=3,height=4];
0->0 [label="Red",comment="Heat",width=1,height=2];
0->1 [label="Orange",comment="Lose$$$$SPACE$$$heat",width=3,height=4];
1->0 [label="Yellow$$$$SPACE$$$cold",comment="Heat",width=5,height=6];
1->1 [label="Green$$$$SPACE$$$cold",comment="Stay$$$$SPACE$$$cool",width=7,height=8];
}
```

---



### 6.6.5 The .svg file produced

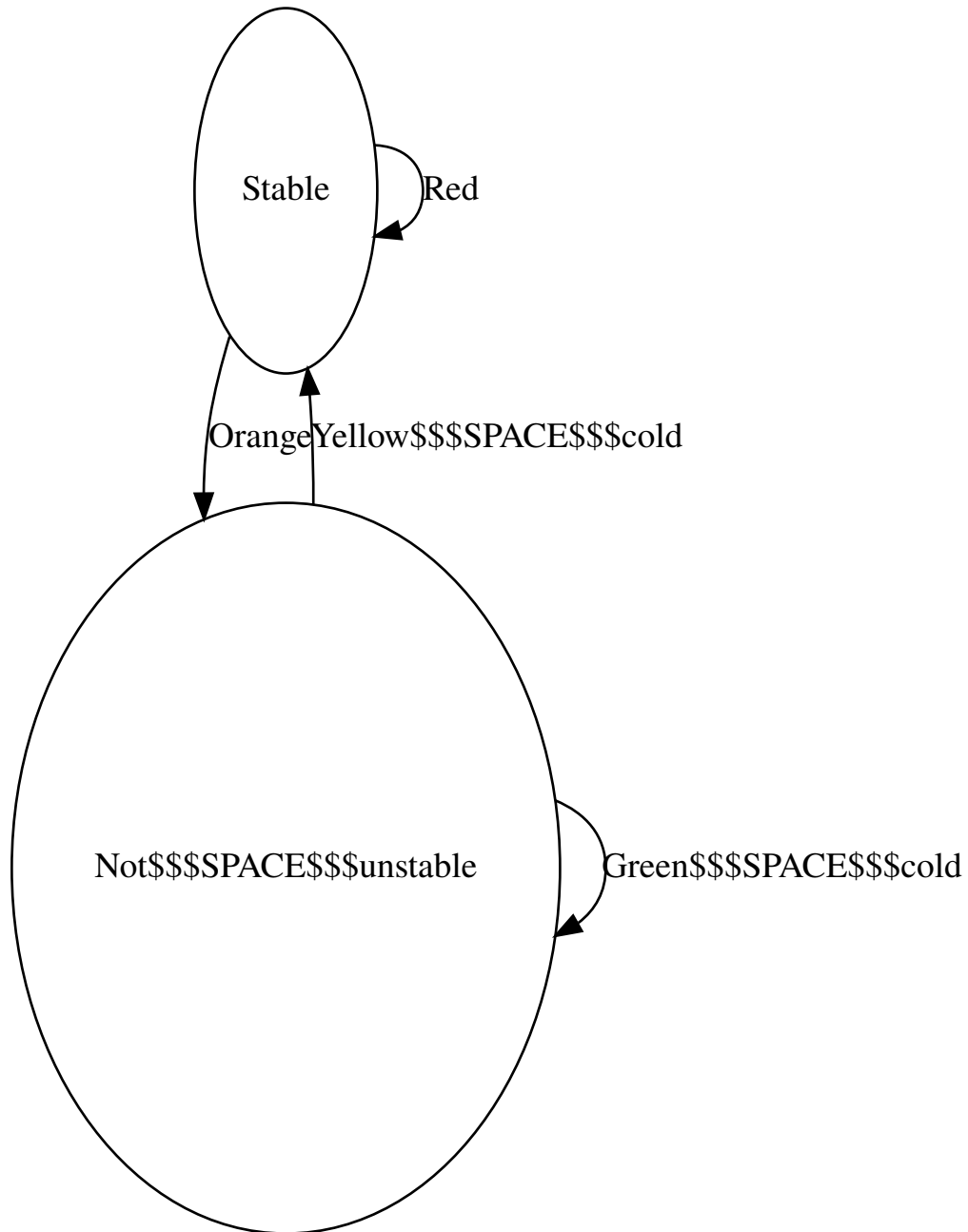


Figure 23: .svg file created from the 'create\_bundled\_edges\_and\_vertices\_markov\_chain' function (algorithm ??) its .dot file, converted from .dot file to .svg using algorithm 136

## 6.7 Creating $K_3$ with bundled edges and vertices

Instead of using edges with a name, or other properties, here we use a bundled edge class called 'my\_bundled\_edge'.

### 6.7.1 Graph

We reproduce the  $K_3$  with named edges and vertices of chapter ?? , but with our bundled edges and vertices instead:

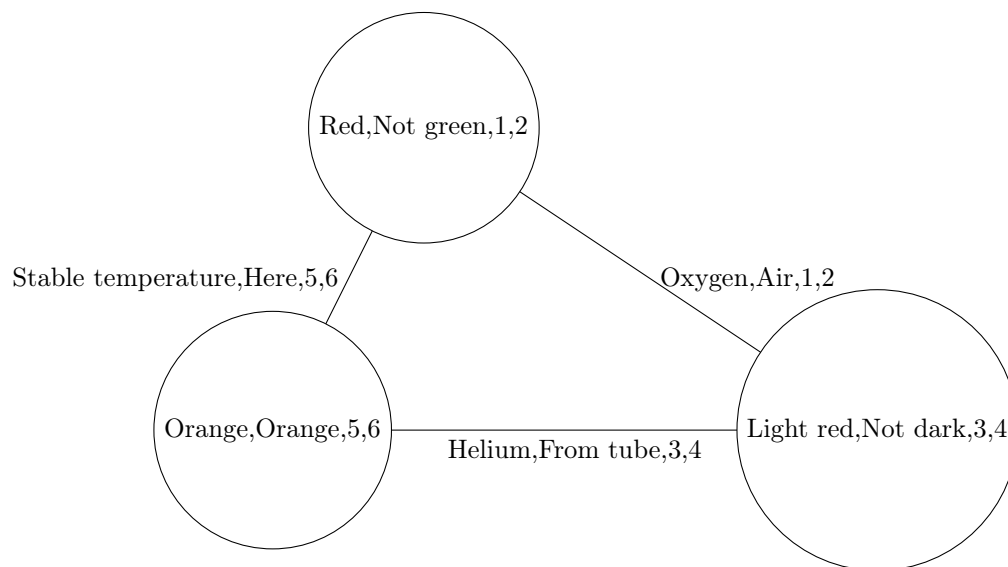


Figure 24:  $K_3$ : a fully connected graph with three named edges and vertices



## 6.7.2 Function to create such a graph

---

**Algorithm 99** Creating  $K_3$  as depicted in figure ??

---

```
#include "
    create_empty_undirected_bundled_edges_and_vertices_graph
    .h"
#include "add_bundled_edge.h"
#include "add_bundled_vertex.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    my_bundled_vertex,
    my_bundled_edge
>
create_bundled_edges_and_vertices_k3_graph()
{
    auto g
        =
            create_empty_undirected_bundled_edges_and_vertices_graph
            ();
    const auto vd_a = add_bundled_vertex(
        my_bundled_vertex("Red", "Not_green", 1.0, 2.0),
        g
    );
    const auto vd_b = add_bundled_vertex(
        my_bundled_vertex("Light_red", "Not_dark", 3.0, 4.0),
        g
    );
    const auto vd_c = add_bundled_vertex(
        my_bundled_vertex("Orange", "Orangy", 5.0, 6.0),
        g
    );
    add_bundled_edge(vd_a, vd_b,
        my_bundled_edge("Oxygen", "Air", 1.0, 2.0),
        g
    );
    add_bundled_edge(vd_b, vd_c,
        my_bundled_edge("Helium", "From_tube", 3.0, 4.0),
        g
    );
    add_bundled_edge(vd_c, vd_a,
        my_bundled_edge("Stable_temperature", "Here", 5.0, 6.0),
        g
    );
    return g;
}
```

---

Most of the code is a slight modification of algorithm ???. In the end, the `my_edges` and `my_vertices` are obtained as the graph its `property_map` and set with the `'my_bundled_edge'` and `'my_bundled_vertex'` objects.

### 6.7.3 Creating such a graph

Here is the demo:

---

**Algorithm 100** Demo of the `'create_bundled_edges_and_vertices_k3_graph'` function (algorithm 99)

---

```
#include <boost/test/unit_test.hpp>
#include "create_bundled_edges_and_vertices_k3_graph.h"

BOOST_AUTO_TEST_CASE(
    test_create_bundled_edges_and_vertices_k3_graph)
{
    auto g
        = create_bundled_edges_and_vertices_k3_graph();
    BOOST_CHECK(boost::num_edges(g) == 3);
    BOOST_CHECK(boost::num_vertices(g) == 3);
}
```

---

### 6.7.4 The .dot file produced

---

**Algorithm 101** .dot file created from the `'create_bundled_edges_and_vertices_markov_chain'` function (algorithm 99), converted from graph to .dot file using algorithm 55

---

```
graph G {
0[label="Red",comment="Not$$$SPACE$$$green",width=1,height=2];
1[label="Light$$$SPACE$$$red",comment="Not$$$SPACE$$$dark",width=3,height=4];
2[label="Orange",comment="Orangy",width=5,height=6];
0--1 [label="Oxygen",comment="Air",width=1,height=2];
1--2 [label="Helium",comment="From$$$SPACE$$$tube",width=3,height=4];
2--0 [label="Stable$$$SPACE$$$temperature",comment="Here",width=5,height=6];
}
```

---





### 6.7.5 The .svg file produced

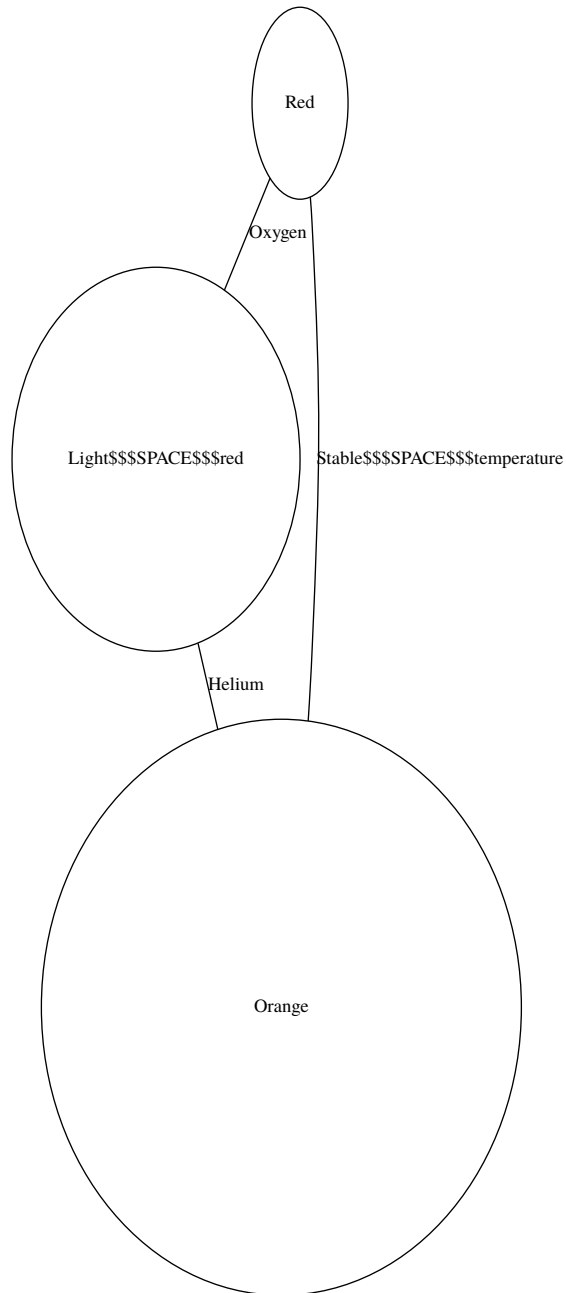


Figure 25: .svg file created from the 'create\_bundled\_edges\_and\_vertices\_k3\_graph' function (algorithm ??) its .dot file, converted from .dot file to .svg using algorithm 136

## 7 Working on graphs with bundled edges and vertices

### 7.1 Has a my\_bundled\_edge

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

---

**Algorithm 102** Find if there is a bundled edge with a certain my\_bundled\_edge

---

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

template <typename graph>
bool has_bundled_edge_with_my_edge(
    const my_bundled_edge& e,
    const graph& g
) noexcept
{
    using ed = typename boost::graph_traits<graph>::
        edge_descriptor;
    const auto eip = edges(g);
    return std::find_if(eip.first, eip.second,
        [&e, &g](const ed& d)
        {
            return g[d] == e;
        }) != eip.second;
}
```

---

This function can be demonstrated as in algorithm 103, where a certain 'my\_bundled\_edge' cannot be found in an empty graph. After adding the desired my\_bundled\_edge, it is found.

---

**Algorithm 103** Demonstration of the ‘has\_bundled\_edge\_with\_my\_edge’ function

---

```
#include <boost/test/unit_test.hpp>
#include "create_bundled_edges_and_vertices_k3_graph.h"
#include "has_bundled_edge_with_my_edge.h"

BOOST_AUTO_TEST_CASE(test_has_bundled_edge_with_my_edge)
{
    auto g
        = create_bundled_edges_and_vertices_k3_graph();
    BOOST_CHECK(
        has_bundled_edge_with_my_edge(
            my_bundled_edge("Oxygen", "Air", 1.0, 2.0), g
        )
    );
}
```

---

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

## 7.2 Find a my\_bundled\_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 104 shows how to obtain an edge descriptor to the first edge found with a specific my\_bundled\_edge value.

---

**Algorithm 104** Find the first bundled edge with a certain `my_bundled_edge`

---

```
#include <cassert>
#include <boost/graph/graph_traits.hpp>
#include "has_bundled_edge_with_my_edge.h"
#include "has_custom_edge_with_my_edge.h"
#include "my_bundled_edge.h"

template <typename graph>
typename boost::graph_traits<graph>::edge_descriptor
find_first_bundled_edge_with_my_edge(
    const my_bundled_edge& e,
    const graph& g
)
{
    using ed = typename boost::graph_traits<graph>::
        edge_descriptor;
    const auto eip = edges(g);
    const auto i = std::find_if(
        eip.first, eip.second,
        [&e, &g](const ed d) { return g[d] == e; }
    );
    if (i == eip.second)
    {
        std::stringstream msg;
        msg << __func__ << ":_ "
            << "could_not_find_my_bundled_edge_"
            << e << "' "
            ;
        throw std::invalid_argument(msg.str());
    }
    return *i;
}
```

---

With the edge descriptor obtained, one can read and modify the edge and the vertices surrounding it. Algorithm 105 shows some examples of how to do so.

---

**Algorithm 105** Demonstration of the ‘find\_first\_bundled\_edge\_with\_my\_edge’ function

---

```
#include <boost/test/unit_test.hpp>
#include "create_bundled_edges_and_vertices_k3_graph.h"
#include "find_first_bundled_edge_with_my_edge.h"

BOOST_AUTO_TEST_CASE(
    test_find_first_bundled_edge_with_my_edge)
{
    const auto g
        = create_bundled_edges_and_vertices_k3_graph();
    const auto ed
        = find_first_bundled_edge_with_my_edge(
            my_bundled_edge("Oxygen", "Air", 1.0, 2.0),
            g
        );
    BOOST_CHECK(boost::source(ed, g)
        != boost::target(ed, g)
    );
}
```

---

### 7.3 Get an edge its my\_bundled\_edge

To obtain the my\_bundled\_edge from an edge descriptor, one needs to pull out the my\_bundled\_edges map and then look up the my\_edge of interest.

---

**Algorithm 106** Get a vertex its my\_bundled\_vertex from its vertex descriptor

---

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

template <typename graph>
my_bundled_edge get_my_bundled_edge(
    const typename boost::graph_traits<graph>::
        edge_descriptor& ed,
    const graph& g
) noexcept
{
    return g[ed];
}
```

---

To use ‘get\_my\_bundled\_edge’, one first needs to obtain an edge descriptor. Algorithm 107 shows a simple example.

---

**Algorithm 107** Demonstration if the ‘get\_my\_bundled\_edge’ function

---

```
#include <boost/test/unit_test.hpp>
#include "add_bundled_edge.h"
#include "add_bundled_vertex.h"
#include "
    create_empty_undirected_bundled_edges_and_vertices_graph
    .h"
#include "find_first_bundled_edge_with_my_edge.h"
#include "get_my_bundled_edge.h"

BOOST_AUTO_TEST_CASE(test_get_my_bundled_edge)
{
    auto g
        =
            create_empty_undirected_bundled_edges_and_vertices_graph
            ();
    const my_bundled_edge edge{"Dex"};
    const auto vd_a = add_bundled_vertex(
        my_bundled_vertex("A"), g
    );
    const auto vd_b = add_bundled_vertex(
        my_bundled_vertex("B"), g
    );
    add_bundled_edge(vd_a, vd_b, edge, g);
    const auto ed
        = find_first_bundled_edge_with_my_edge(edge, g);
    BOOST_CHECK(get_my_bundled_edge(ed, g) == edge);
}
```

---

## 7.4 Set an edge its my\_bundled\_edge

If you know how to get the my\_bundled\_edge from an edge descriptor, setting it is just as easy, as shown in algorithm 108.

---

**Algorithm 108** Set a bundled edge its `my_bundled_edge` from its edge descriptor

---

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

template <typename graph>
void set_my_bundled_edge(
    const my_bundled_edge& name,
    const typename boost::graph_traits<graph>::
        edge_descriptor& ed,
    graph& g
) noexcept
{
    static_assert(!std::is_const<graph>::value,
        "graph_cannot_be_const");
    g[ed] = name;
}
```

---

To use ‘`set_bundled_edge_my_edge`’, one first needs to obtain an edge descriptor. Algorithm 109 shows a simple example.

---

**Algorithm 109** Demonstration if the ‘set\_bundled\_edge\_my\_edge’ function

---

```
#include <boost/test/unit_test.hpp>
#include "add_bundled_edge.h"
#include "add_bundled_vertex.h"
#include "
    create_empty_undirected_bundled_edges_and_vertices_graph
    .h"
#include "find_first_bundled_edge_with_my_edge.h"
#include "get_my_bundled_edge.h"
#include "set_my_bundled_edge.h"

BOOST_AUTO_TEST_CASE(test_set_my_bundled_edge)
{
    auto g
        =
            create_empty_undirected_bundled_edges_and_vertices_graph
            ();
    const auto vd_a = add_bundled_vertex(my_bundled_vertex{
        "A"}, g);
    const auto vd_b = add_bundled_vertex(my_bundled_vertex{
        "B"}, g);
    const my_bundled_edge old_edge{"Dex"};
    add_bundled_edge(vd_a, vd_b, old_edge, g);
    const auto vd
        = find_first_bundled_edge_with_my_edge(old_edge, g);
    BOOST_CHECK(get_my_bundled_edge(vd, g)
        == old_edge
    );
    const my_bundled_edge new_edge{"Diggy"};
    set_my_bundled_edge(new_edge, vd, g);
    BOOST_CHECK(get_my_bundled_edge(vd, g)
        == new_edge
    );
}
```

---

## 7.5 Storing a graph with bundled edges and vertices as a .dot

If you used the ‘create\_bundled\_edges\_and\_vertices\_k3\_graph’ function (algorithm 99) to produce a  $K_3$ graph with edges and vertices associated with my\_bundled\_edge and my\_bundled\_vertex objects, you can store these my\_bundled\_edges and my\_bundled\_vertex-es additionally with algorithm 110:



---

**Algorithm 110** Storing a graph with bundled edges and vertices as a .dot file

---

```
#include <fstream>
#include <boost/graph/graphviz.hpp>
#include "make_bundled_vertices_writer.h"
#include "make_bundled_edges_writer.h"

template <typename graph>
void save_bundled_edges_and_vertices_graph_to_dot(
    const graph& g,
    const std::string& filename
)
{
    std::ofstream f(filename);
    boost::write_graphviz(
        f,
        g,
        make_bundled_vertices_writer(g),
        make_bundled_edges_writer(g)
    );
}
```

---

## 7.6 Load a directed graph with bundled edges and vertices from a .dot file

When loading a graph from file, one needs to specify a type of graph. In this example, an directed graph with bundled edges and vertices is loaded, as shown in algorithm 111:

---

**Algorithm 111** Loading a directed graph with bundled edges and vertices from a .dot file

---

```

#include <fstream>
#include <boost/graph/graphviz.hpp>
#include "
    create_empty_directed_bundled_edges_and_vertices_graph
    .h"
#include "is_regular_file.h"
#include "graphviz_decode.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::directedS,
    my_bundled_vertex,
    my_bundled_edge
>
load_directed_bundled_edges_and_vertices_graph_from_dot(
    const std::string& dot_filename
)
{
    if (!is_regular_file(dot_filename))
    {
        std::stringstream msg;
        msg << __func__ << ":_file_"
            << dot_filename << " '_not_found"
            ;
        throw std::invalid_argument(msg.str());
    }
    std::ifstream f(dot_filename.c_str());
    auto g =
        create_empty_directed_bundled_edges_and_vertices_graph
        ();

    boost::dynamic_properties dp(boost::
        ignore_other_properties);
    dp.property("label", get(&my_bundled_vertex::m_name, g))
        ;
    dp.property("comment", get(&my_bundled_vertex::
        m_description, g));
    dp.property("width", get(&my_bundled_vertex::m_x, g));
    dp.property("height", get(&my_bundled_vertex::m_y, g));
    dp.property("edge_id", get(&my_bundled_edge::m_name, g))
        ;
    dp.property("label", get(&my_bundled_edge::m_name, g));
    dp.property("comment", get(&my_bundled_edge::
        m_description, g));
    dp.property("width", get(&my_bundled_edge::m_width, g))
        ;
    dp.property("height", get(&my_bundled_edge::m_height, g
        ));
    boost::read_graphviz(f, g, dp);

    //Decode vertices
    {

```

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 its default constructor, after which we direct the `boost::dynamic_properties` to find a `'node_id'` and `'label'` in the vertex name map, `'edge_id'` and `'label'` to the edge name map. From this and the empty graph, `'boost::read_graphviz'` is called to build up the graph.

Algorithm 112 shows how to use the `'load_directed_bundled_edges_and_vertices_graph_from_dot'` function:

---

**Algorithm 112** Demonstration of the ‘load\_directed\_bundled\_edges\_and\_vertices\_graph\_from\_dot’function

---

```
#include <boost/test/unit_test.hpp>
#include "create_bundled_edges_and_vertices_markov_chain.h"
#include "get_sorted_bundled_vertex_my_vertexes.h"
#include "load_directed_bundled_edges_and_vertices_graph_from_dot.h"
#include "save_bundled_edges_and_vertices_graph_to_dot.h"

BOOST_AUTO_TEST_CASE(
    test_load_directed_bundled_edges_and_vertices_graph_from_dot
)
{
    using boost::num_edges;
    using boost::num_vertices;

    const auto g
        = create_bundled_edges_and_vertices_markov_chain();
    const std::string filename{
        "create_bundled_edges_and_vertices_markov_chain.dot"
    };
    save_bundled_edges_and_vertices_graph_to_dot(g,
        filename);
    const auto h
        =
        load_directed_bundled_edges_and_vertices_graph_from_dot
        (
            filename
        );
    BOOST_CHECK(num_edges(g) == num_edges(h));
    BOOST_CHECK(num_vertices(g) == num_vertices(h));
    BOOST_CHECK(get_sorted_bundled_vertex_my_vertexes(g)
        == get_sorted_bundled_vertex_my_vertexes(h)
    );
}
```

---

This demonstration shows how the Markov chain is created using the ‘create\_bundled\_edges\_and\_vertices\_markov\_chain’ function (algorithm 96), saved and then loaded.

### **7.7 Load an undirected graph with bundled edges and vertices from a .dot file**

When loading a graph from file, one needs to specify a type of graph. In this example, an undirected graph with bundled edges and vertices is loaded, as shown in algorithm 113:

---

**Algorithm 113** Loading an undirected graph with bundled edges and vertices from a .dot file

---

```

#include <fstream>
#include <boost/graph/graphviz.hpp>
#include "
    create_empty_undirected_bundled_edges_and_vertices_graph
    .h"
#include "is_regular_file.h"
#include "graphviz_decode.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    my_bundled_vertex,
    my_bundled_edge
>
load_undirected_bundled_edges_and_vertices_graph_from_dot
(
    const std::string& dot_filename
)
{
    if (!is_regular_file(dot_filename))
    {
        std::stringstream msg;
        msg << __func__ << ":_file_"
            << dot_filename << ":_not_found"
        ;
        throw std::invalid_argument(msg.str());
    }
    std::ifstream f(dot_filename.c_str());
    auto g =
        create_empty_undirected_bundled_edges_and_vertices_graph
        ();

    boost::dynamic_properties dp(boost::
        ignore_other_properties);
    dp.property("label", get(&my_bundled_vertex::m_name, g))
    ;
    dp.property("comment", get(&my_bundled_vertex::
        m_description, g));
    dp.property("width", get(&my_bundled_vertex::m_x, g));
    dp.property("height", get(&my_bundled_vertex::m_y, g));
    dp.property("edge_id", get(&my_bundled_edge::m_name, g))
    ;
    dp.property("label", get(&my_bundled_edge::m_name, g));
    dp.property("comment", get(&my_bundled_edge::
        m_description, g));
    dp.property("width", get(&my_bundled_edge::m_width, g))
    ;
    dp.property("height", get(&my_bundled_edge::m_height, g
    ));
    boost::read_graphviz(f, g, dp);

    //Decode vertices

```

The only difference with loading a directed graph, is that the initial empty graph is undirected instead. Chapter 7.6 describes the rationale of this function.

Algorithm 114 shows how to use the ‘load\_undirected\_bundled\_vertices\_graph\_from\_dot’ function:

---

**Algorithm 114** Demonstration of the ‘load\_undirected\_bundled\_edges\_and\_vertices\_graph\_from\_dot’fun

---

```
#include <boost/test/unit_test.hpp>
#include "create_bundled_edges_and_vertices_k3_graph.h"
#include "get_sorted_bundled_vertex_my_vertexes.h"
#include "
    load_undirected_bundled_edges_and_vertices_graph_from_dot
    .h"
#include "save_bundled_edges_and_vertices_graph_to_dot.h"

BOOST_AUTO_TEST_CASE(
    test_load_undirected_bundled_edges_and_vertices_graph_from_dot
)
{
    using boost::num_edges;
    using boost::num_vertices;

    const auto g
        = create_bundled_edges_and_vertices_k3_graph();
    const std::string filename{
        "create_bundled_edges_and_vertices_k3_graph.dot"
    };
    save_bundled_edges_and_vertices_graph_to_dot(g,
        filename);
    const auto h
        =
        load_undirected_bundled_edges_and_vertices_graph_from_dot
        (
            filename
        );
    BOOST_CHECK(num_edges(g) == num_edges(h));
    BOOST_CHECK(num_vertices(g) == num_vertices(h));
    BOOST_CHECK(get_sorted_bundled_vertex_my_vertexes(g)
        == get_sorted_bundled_vertex_my_vertexes(h)
    );
}
```

---

This demonstration shows how  $K_2$  with bundled vertices is created using the ‘create\_bundled\_vertices\_k2\_graph’ function (algorithm ??), saved and then loaded. The loaded graph is checked to be a graph similar to the original.

## 8 Building graphs with a graph name

Up until now, the graphs created have had no properties themselves. Sure, the edges and vertices have had properties, but the graph itself has had none. Until now.

In this chapter, graphs will be created with a graph name of type `std::string`

- An empty directed graph with a graph name: see chapter
- An empty undirected graph with a graph name: see chapter
- A two-state Markov chain with a graph name: see chapter
- $K_3$  with a graph name: see chapter

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

- Getting a graph its name: see chapter
- Setting a graph its name: see chapter

### 8.1 Create an empty directed graph with a graph name property

Algorithm 115 shows the function to create an empty directed graph with a graph name.

---

**Algorithm 115** Creating an empty directed graph with a graph name

---

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

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

---

This `boost::adjacency_list` is of the following type:



- the first ‘boost::vecS’: select (that is what the ‘S’ means) that out edges are stored in a std::vector. This is the default way.
- the second ‘boost::vecS’: select that the graph vertices are stored in a std::vector. This is the default way.
- ‘boost::directedS’: select that the graph is directed. This is the default selectedness
- the first ‘boost::no\_property’: the vertices have no properties. This is the default (non-)property
- the second ‘boost::no\_property’: the vertices have no properties. This is the default (non-)property
- ‘boost::property<boost::graph\_name\_t, std::string>’: the graph itself has a single property: its boost::graph\_name has type std::string

Algorithm 116 demonstrates the ‘create\_empty\_directed\_graph\_with\_graph\_name’ function.

---

**Algorithm 116** Demonstration of ‘create\_empty\_directed\_graph\_with\_graph\_name’

---

```
#include <boost/test/unit_test.hpp>
#include "create_empty_directed_graph_with_graph_name.h"

BOOST_AUTO_TEST_CASE(
    test_create_empty_directed_graph_with_graph_name)
{
    auto g
        = create_empty_directed_graph_with_graph_name();
    BOOST_CHECK(boost::num_edges(g) == 0);
    BOOST_CHECK(boost::num_vertices(g) == 0);
}
```

---

## 8.2 Create an empty undirected graph with a graph name property

Algorithm 117 shows the function to create an empty undirected graph with a graph name.

---

**Algorithm 117** Creating an empty undirected 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_undirected_graph_with_graph_name() noexcept
{
    return {};
}
```

---

This code is very similar to the code described in chapter 115, except that the directness (the third template argument) is undirected (due to the `boost::undirectedS`).

Algorithm 118 demonstrates the ‘`create_empty_undirected_graph_with_graph_name`’ function.

---

**Algorithm 118** Demonstration of ‘`create_empty_undirected_graph_with_graph_name`’

---

```
#include <boost/test/unit_test.hpp>
#include "create_empty_undirected_graph_with_graph_name.h"

BOOST_AUTO_TEST_CASE(
    test_create_empty_undirected_graph_with_graph_name)
{
    auto g = create_empty_undirected_graph_with_graph_name
        ();
    BOOST_CHECK(boost::num_edges(g) == 0);
    BOOST_CHECK(boost::num_vertices(g) == 0);
}
```

---

### 8.3 Get a graph its name property

---

**Algorithm 119** 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_property(
        g, boost::graph_name
    );
}
```

---

Algorithm 120 demonstrates the ‘get\_graph\_name’ function.

---

**Algorithm 120** Demonstration of ‘get\_graph\_name’

---

```
#include <boost/test/unit_test.hpp>
#include "create_empty_directed_graph_with_graph_name.h"
#include "get_graph_name.h"
#include "set_graph_name.h"

BOOST_AUTO_TEST_CASE(test_get_graph_name)
{
    auto g = create_empty_directed_graph_with_graph_name();
    const std::string name{"Dex"};
    set_graph_name(name, g);
    BOOST_CHECK(get_graph_name(g) == name);
}
```

---

## 8.4 Set a graph its name property

---

**Algorithm 121** 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
{
    static_assert(!std::is_const<graph>::value,
        "graph_cannot_be_const"
    );
    get_property(g, boost::graph_name) = name;
}
```

---

Algorithm 122 demonstrates the ‘set\_graph\_name’ function.

---

**Algorithm 122** Demonstration of ‘set\_graph\_name’

---

```
#include <boost/test/unit_test.hpp>
#include "create_empty_directed_graph_with_graph_name.h"
#include "get_graph_name.h"
#include "set_graph_name.h"

BOOST_AUTO_TEST_CASE(test_set_graph_name)
{
    auto g = create_empty_directed_graph_with_graph_name();
    const std::string name{"Dex"};
    set_graph_name(name, g);
    BOOST_CHECK(get_graph_name(g) == name);
}
```

---

## 8.5 Create a directed graph with a graph name property

### 8.5.1 Graph

See figure 5.

### 8.5.2 Function to create such a graph

Algorithm 123 shows the function to create an empty directed graph with a graph name.

---

**Algorithm 123** Creating a two-state Markov chain with a graph name

---

```
#include <cassert>
#include "create_empty_directed_graph_with_graph_name.h"
#include "set_graph_name.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::directedS,
    boost::no_property,
    boost::no_property,
    boost::property<boost::graph_name_t, std::string>
>
create_markov_chain_with_graph_name() noexcept
{
    auto g = create_empty_directed_graph_with_graph_name();
    const auto vd_a = boost::add_vertex(g);
    const auto vd_b = boost::add_vertex(g);
    boost::add_edge(vd_a, vd_a, g);
    boost::add_edge(vd_a, vd_b, g);
    boost::add_edge(vd_b, vd_a, g);
    boost::add_edge(vd_b, vd_b, g);

    set_graph_name("Two-state_Markov_chain", g);
    return g;
}
```

---

### 8.5.3 Creating such a graph

Algorithm 124 demonstrates the ‘create\_markov\_chain\_with\_graph\_name’ function.

---

**Algorithm 124** Demonstration of ‘create\_markov\_chain\_with\_graph\_name’

---

```
#include <boost/test/unit_test.hpp>
#include "create_markov_chain_with_graph_name.h"
#include "get_graph_name.h"

BOOST_AUTO_TEST_CASE(
    test_create_markov_chain_with_graph_name)
{
    const auto g = create_markov_chain_with_graph_name();
    BOOST_CHECK(boost::num_vertices(g) == 2);
    BOOST_CHECK(boost::num_edges(g) == 4);
    BOOST_CHECK(get_graph_name(g) == "Two-state_Markov_
        chain");
}
```

---

#### 8.5.4 The .dot file produced

---

**Algorithm 125** .dot file created from the ‘create\_markov\_chain\_with\_graph\_name’ function (algorithm 123), converted from graph to .dot file using algorithm 55

---

```
digraph G {
name="Two-state Markov chain";
0;
1;
0->0 ;
0->1 ;
1->0 ;
1->1 ;
}
```

---

### 8.5.5 The .svg file produced

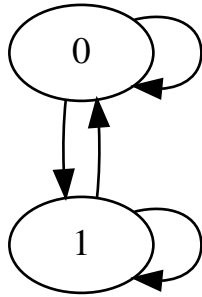


Figure 26: .svg file created from the ‘create\_markov\_chain\_with\_graph\_name’ function (algorithm 123) its .dot file, converted from .dot file to .svg using algorithm 136

## 8.6 Create an undirected graph with a graph name property

### 8.6.1 Graph

See figure 7.

### 8.6.2 Function to create such a graph

Algorithm 126 shows the function to create K2 graph with a graph name.

---

**Algorithm 126** Creating a K2 graph with a graph name

---

```
#include "create_empty_undirected_graph_with_graph_name.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    boost::no_property,
    boost::no_property,
    boost::property<boost::graph_name_t, std::string>
>
create_k2_graph_with_graph_name() noexcept
{
    auto g = create_empty_undirected_graph_with_graph_name
        ();
    const auto vd_a = boost::add_vertex(g);
    const auto vd_b = boost::add_vertex(g);
    boost::add_edge(vd_a, vd_b, g);
    get_property(g, boost::graph_name) = "K2";

    return g;
}
```

---

### 8.6.3 Creating such a graph

Algorithm 127 demonstrates the ‘create\_k2\_graph\_with\_graph\_name’ function.

---

**Algorithm 127** Demonstration of ‘create\_k2\_graph\_with\_graph\_name’

---

```
#include <boost/test/unit_test.hpp>
#include "create_k2_graph_with_graph_name.h"
#include "get_graph_name.h"

BOOST_AUTO_TEST_CASE(test_create_k2_graph_with_graph_name)
{
    const auto g = create_k2_graph_with_graph_name();
    BOOST_CHECK(boost::num_vertices(g) == 2);
    BOOST_CHECK(boost::num_edges(g) == 1);
    BOOST_CHECK(get_graph_name(g) == "K2");
}
```

---



#### 8.6.4 The .dot file produced

---

**Algorithm 128** .dot file created from the ‘create\_k2\_graph\_with\_graph\_name’ function (algorithm 126), converted from graph to .dot file using algorithm 55

---

```
graph G {  
  name="K2";  
  0;  
  1;  
  0--1 ;  
}
```

---

#### 8.6.5 The .svg file produced

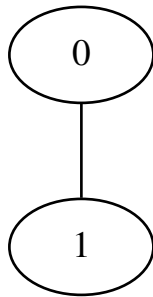


Figure 27: .svg file created from the ‘create\_k2\_graph\_with\_graph\_name’ function (algorithm 126) its .dot file, converted from .dot file to .svg using algorithm 136

## 9 Working on graphs with a graph name

### 9.1 Storing a graph with a graph name property as a .dot file

This works:

---

**Algorithm 129** Storing a graph with a graph name as a .dot file

---

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

#include "get_graph_name.h"

template <typename graph>
void save_graph_with_graph_name_to_dot(
    const graph& g,
    const std::string& filename
)
{
    std::ofstream f(filename);
    boost::write_graphviz(
        f,
        g,
        boost::default_writer(),
        boost::default_writer(),
        //Unsure if this results in a graph
        //that can be loaded correctly
        //from a .dot file
        [&g](std::ostream& os) {
            os << "name=\""
                << get_graph_name(g)
                << "\";\n";
        }
    );
}
```

---

## 9.2 Loading a directed graph with a graph name property from a .dot file

This will result in a directed graph with a name:

---

**Algorithm 130** Loading a directed graph with a graph name from a .dot file

---

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

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::directedS,
    boost::no_property,
    boost::no_property,
    boost::property<
        boost::graph_name_t, std::string
    >
>
>
load_directed_graph_with_graph_name_from_dot(
    const std::string& dot_filename
)
{
    using graph = boost::adjacency_list<
        boost::vecS,
        boost::vecS,
        boost::directedS,
        boost::no_property,
        boost::no_property,
        boost::property<
            boost::graph_name_t, std::string
        >
    >
>;

    if (!is_regular_file(dot_filename))
    {
        std::stringstream msg;
        msg << __func__ << ":_file_"
            << dot_filename << " '_not_found"
        ;
        throw std::invalid_argument(msg.str());
    }

    graph g;

    boost::ref_property_map<graph*, std::string>
    graph_name{
        get_property(g, boost::graph_name)
    };
    boost::dynamic_properties dp{
        boost::ignore_other_properties
    };
    dp.property("name", graph_name);

    std::ifstream f(dot_filename.c_str());
    boost::read_graphviz(f, g, dp);
    return g;
}
```

### **9.3 Loading an undirected graph with a graph name property from a .dot file**

This will result in an undirected graph with a name:

---

**Algorithm 131** Loading an undirected graph with a graph name from a .dot file

---

```

#include <fstream>
#include <string>
#include <boost/graph/graphviz.hpp>
#include "create_empty_undirected_graph_with_graph_name.h"
"
#include "is_regular_file.h"

boost::adjacency_list<
    boost::vecS,
    boost::vecS,
    boost::undirectedS,
    boost::no_property,
    boost::no_property,
    boost::property<
        boost::graph_name_t, std::string
    >
>
>
load_undirected_graph_with_graph_name_from_dot(
    const std::string& dot_filename
)
{
    using graph = boost::adjacency_list<
        boost::vecS,
        boost::vecS,
        boost::undirectedS,
        boost::no_property,
        boost::no_property,
        boost::property<
            boost::graph_name_t, std::string
        >
    >
    >;
    if (!is_regular_file(dot_filename))
    {
        std::stringstream msg;
        msg << __func__ << ":'_file_' "
            << dot_filename << ":'_not_found"
        ;
        throw std::invalid_argument(msg.str());
    }
    graph g;

    boost::ref_property_map<graph*, std::string>
    graph_name{
        get_property(g, boost::graph_name)
    };
    boost::dynamic_properties dp{
        boost::ignore_other_properties
    };
    dp.property("name", graph_name);

    std::ifstream f(dot_filename.c_str());
    boost::read_graphviz(f, g, dp);
    return g;
}

```

## 10 Other graph functions

Some functions that did not fit in.

### 10.1 Encode a `std::string` to a Graphviz-friendly format

You may want to use a label with spaces, comma's and/or quotes. Saving and loading these, will result in problem. This function replaces these special characters by a rare combination of ordinary characters.

---

**Algorithm 132** Encode a `std::string` to a Graphviz-friendly format

---

```
#include <boost/algorithm/string/replace.hpp>

std::string graphviz_encode(std::string s) noexcept
{
    boost::algorithm::replace_all(s, ",", "$$$COMMA$$$");
    boost::algorithm::replace_all(s, "_", "$$$SPACE$$$");
    boost::algorithm::replace_all(s, "\"", "$$$QUOTE$$$");
    return s;
}
```

---

### 10.2 Decode a `std::string` from a Graphviz-friendly format

This function undoes the ‘graphviz\_encode’ function (algorithm 132) and thus converts a Graphviz-friendly `std::string` to the original human-friendly `std::string`.

---

**Algorithm 133** Decode a `std::string` from a Graphviz-friendly format to a human-friendly format

---

```
#include <boost/algorithm/string/replace.hpp>

std::string graphviz_decode(std::string s) noexcept
{
    boost::algorithm::replace_all(s, "$$$COMMA$$$", ",");
    boost::algorithm::replace_all(s, "$$$SPACE$$$", "_");
    boost::algorithm::replace_all(s, "$$$QUOTE$$$", "\"");
    return s;
}
```

---

### 10.3 Check if a `std::string` is Graphviz-friendly

There are pieces where I check if a `std::string` is Graphviz-friendly. This is done only where it matters. If it is tested not to matter, ‘is\_graphviz\_friendly’ is absent.

---

**Algorithm 134** Check if a `std::string` is Graphviz-friendly

---

```
#include "graphviz_encode.h"

bool is_graphviz_friendly(const std::string& s) noexcept
{
    return graphviz_encode(s) == s;
}
```

---

## 11 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, platform-dependent and/or there may be superior alternatives. I just add them for completeness.

### 11.1 Getting a data type as a `std::string`

This function will only work under GCC. I found this code at: <http://stackoverflow.com/questions/1055452/c-get-name-of-type-in-template>. Thanks to 'm-dudley' (Stack Overflow user page at <http://stackoverflow.com/users/111327/m-dudley>).

---

**Algorithm 135** Getting a data type its name as a `std::string`

---

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

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;
}
```

---

## 11.2 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 136** Convert a .dot file to a .svg

---

```
#include <cassert>
#include <string>
#include <iostream>
#include <sstream>
#include <stdexcept>
#include "has_dot.h"
#include "is_regular_file.h"
#include "is_valid_dot_file.h"

void convert_dot_to_svg(
    const std::string& dot_filename,
    const std::string& svg_filename
)
{
    if (!has_dot())
    {
        std::stringstream msg;
        msg << __func__ << ": 'dot' cannot be found."
            << "type 'sudo apt-get install graphviz' in the
                command_line"
        ;
        throw std::runtime_error(msg.str());
    }
    if (!is_valid_dot_file(dot_filename))
    {
        std::stringstream msg;
        msg << __func__ << ": 'file'" << dot_filename
            << "' is not a valid DOT language"
        ;
        throw std::invalid_argument(msg.str());
    }
    std::stringstream cmd;
    cmd << "dot -Tsvg" << dot_filename << "-o" <<
        svg_filename;
    const int error {
        std::system(cmd.str().c_str())
    };
    if (error)
    {
        std::cerr << __func__ << ": warning: command '"
            << cmd.str() << "' resulting in error"
            << error;
    }
    if (!is_regular_file(svg_filename))
    {
        std::stringstream msg;
        msg << __func__ << ": failed to create SVG output
            file '"
            << svg_filename << "'"
        ;
        throw std::runtime_error(msg.str());
    }
}
```

---

‘convert\_dot\_to\_svg’ makes a system call to the program ‘dot’ to convert the .dot file to an .svg file.

### 11.3 Check if a file exists

Not the most smart way perhaps, but it does only use the STL.

---

**Algorithm 137** Check if a file exists

---

```
#include <fstream>

bool is_regular_file(const std::string& filename)
    noexcept
{
    std::fstream f;
    f.open(filename.c_str(), std::ios::in);
    return f.is_open();
}
```

---

## 12 Errors

Some common errors.

### 12.1 Formed reference to void

This compile-time error occurs when you create a graph without a certain property, then subsequently reading that property, as in algorithm 138:

---

**Algorithm 138** 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 138 a graph is created with vertices of no properties. Then the names of these vertices, which do not exist, are tried to be read. If you want to read the names of the vertices, supply a graph that has this property.

## 12.2 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 139** Creating the error ‘no matching function for call to clear\_out\_edges’

---

```
#include "create_k2_graph.h"

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

---

In algorithm 139 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.

## 12.3 No matching function for call to ‘clear\_in\_edges’

See chapter 12.2.

## 12.4 Undefined reference to boost::detail::graph::read\_graphviz\_new

You will have to link against 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
```

## 12.5 Property not found: node\_id

When loading a graph from file (as in chapter 3.13) 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 initialize with ‘boost::ignore\_other\_properties’ as a constructor argument instead. Algorithm 140 shows how to trigger this run-time error.

---

**Algorithm 140** Creating the error ‘Property not found: node\_id’

---

```
#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 dp(boost::
        ignore_other_properties);
    boost::dynamic_properties dp; //Error

    try {
        boost::read_graphviz(f,g,dp);
    }
    catch (std::exception&) {
        return; //Should get here
    }
    assert(!"Should_not_get_here");
}
```

---

## 12.6 Stream zeroes

When loading a graph from a .dot file, in operator>>, I encountered reading zeroes, where I expected an XML formatted string:

```
std::istream& ribi::cmap::operator>>(std::istream& is, my_class& any_class) noexcept
{
    std::string s;
```

```

    is >> s; //s has an XML format
    assert(s != "0");
    any_class = my_class(s);
    return is;
}

```

This was because I misconfigured the reader. I did (heavily simplified code):

```

graph load_from_dot(const std::string& dot_filename)
{
    std::ifstream f(dot_filename.c_str());
    graph g;
    boost::dynamic_properties dp;
    dp.property("node_id", get(boost::vertex_custom_type, g));
    dp.property("label", get(boost::vertex_custom_type, g));
    boost::read_graphviz(f,g,dp);
    return g;
}

```

Where it should have been:

```

graph load_from_dot(const std::string& dot_filename)
{
    std::ifstream f(dot_filename.c_str());
    graph g;
    boost::dynamic_properties dp(boost::ignore_other_properties);
    dp.property("label", get(boost::vertex_custom_type, g));
    boost::read_graphviz(f,g,dp);
    return g;
}

```

The explanation is that by setting the boost::dynamic\_property ‘node\_id’ to ‘boost::vertex\_custom\_type’, operator>> will receive the node indices.

An alternative, but less clean solution, is to let operator>> ignore the node indices:

```

std::istream& ribi::cmap::operator>>(std::istream& is, my_class& any_class) noexcept
{
    std::string s;
    is >> s; //s has an XML format
    if (!is_xml(s)) { //Ignore node index
        any_class_class = my_class();
    }
    else {
        any_class_class = my_class(s);
    }
    return is;
}

```

## 13 Appendix

### 13.1 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

### 13.2 Graphviz attributes

List created from [www.graphviz.org/content/attrs](http://www.graphviz.org/content/attrs), where only the attributes that are supported by all formats are listed:

Edge	Graph	Vertex
arrowhead	_background	color
arrowsize	bgbcolor	colorscheme
arrowtail	center	comment
color	charset	distortion
colorscheme	color	fillcolor
comment	colorscheme	fixedsize
decorate	comment	fontcolor
dir	concentrate	fontname
fillcolor	fillcolor	fontsize
fontcolor	fontcolor	gradientangle
fontname	fontname	height
fontsize	fontpath	image
gradientangle	fontsize	imagescale
headclip	forcelabels	label
headlabel	gradientangle	labelloc
headport	imagepath	layer
label	label	margin
labelangle	labeljust	nojustify
labeldistance	labelloc	orientation
labelfloat	landscape	penwidth
labelfontcolor	layerlistsep	peripheries
labelfontname	layers	pos
labelfontsize	layerselect	regular
layer	layersep	samplepoints
nojustify	layout	shape
penwidth	margin	shapefile
pos	nodesep	sides
style	nojustify	skew
tailclip	orientation	sortv
taillabel	outputorder	style
tailport	pack	width
weight	packmode	xlabel
xlabel	pad	z
	page	
	pagedir	
	penwidth	
	quantum	
	ratio	
	rotate	
	size	
	sortv	
	splines	
	style	
	viewport	

## 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

- #include, 12
- $K_2$ , create, 29
- $K_3$ , create, 32
  
- Add a vertex, 17
- Add an edge, 21
- Add bundled edge, 100
- Add bundled vertex, 67
- Add vertex, 17
- add\_edge, 22
- aer\_, 23
- All edge properties, 150
- All graph properties, 150
- All vertex properties, 150
- assert, 15, 22
- auto, 13
  
- boost::add\_edge, 21, 22, 27, 30
- boost::add\_edge result, 23
- boost::add\_vertex, 17, 27, 30
- boost::adjacency\_list, 13
- boost::adjacency\_matrix, 13
- boost::degree does not exist, 43
- boost::directedS, 14, 66, 129
- boost::dynamic\_properties, 61, 91, 123, 147
- boost::edge does not exist, 45
- boost::edge\_bundled\_type\_t, 97
- boost::edges does not exist, 23–25
- boost::get does not exist, 7
- boost::graph\_name, 129
- boost::graph\_name\_t, 129
- boost::ignore\_other\_properties, 61, 147
- boost::in\_degree does not exist, 43
- boost::no\_property, 129
- boost::num\_edges, 16
- boost::num\_vertices, 15
- boost::out\_degree does not exist, 43
- boost::property, 97, 129
- boost::read\_graphviz, 61, 91, 123, 147
- boost::undirectedS, 14, 67, 98, 130
- boost::vecS, 14, 66, 129
  
- boost::vertices does not exist, 19, 21, 25
- boost::write\_graphviz, 60
- bundled\_vertices\_writer, 88
  
- const, 13
- const-correctness, 13
- Convert dot to svg, 145
- Count connected components, 54, 56
- Count undirected graph levels, 58
- Counting the number of edges, 16
- Counting the number of vertices, 15
- Create  $K_2$ , 29
- Create  $K_2$  graph, 30
- Create  $K_3$ , 32
- Create  $K_3$  graph, 33
- Create .dot from graph, 59
- Create .dot from graph with bundled edges and vertices, 120
- Create all direct-neighbour subgraphs, 51
- Create an empty directed graph, 12
- Create an empty graph, 14
- Create bundled edges and vertices  $K_3$  graph, 110
- Create bundled edges and vertices Markov chain, 104
- Create bundled vertices  $K_2$  graph, 75
- Create bundled vertices Markov chain, 70
- Create direct-neighbour subgraph, 48
- Create direct-neighbour subgraph\_including\_in\_edges, 50
- Create directed graph, 26
- Create directed graph from .dot, 60
- Create empty directed bundled edges and vertices graph, 97
- Create empty directed bundled vertices graph, 66
- Create empty directed graph, 12
- Create empty directed graph with graph name, 128

Create empty undirected bundled edges and vertices graph, 98	Get bundled vertex my_vertexes, 68
Create empty undirected bundled vertices graph, 67	Get edge between vertices, 46
Create empty undirected graph, 14	Get edge descriptors, 25
Create empty undirected graph with graph name, 130	Get edge iterators, 24
Create K2 graph with graph name, 136	Get edge my_bundled_edges, 102
Create Markov chain, 27	Get graph name, 131
Create Markov chain with graph name, 133	Get my_bundled_edge, 117
Create path graph, 34, 35	Get n edges, 16
Create Petersen graph, 37, 39	Get n vertices, 15
Create undirected graph from .dot, 62	Get type name, 144
Create undirected graph with bundled edges and vertices from .dot, 125	Get vertex descriptors, 20
Declaration, my_bundled_edge, 96	Get vertex iterators, 19
Declaration, my_bundled_vertex, 65	Get vertex out degrees, 43
decltype(auto), 7	Get vertices, 19
directed graph, 9	Graph properties, 150
Directed graph, create, 26	Graphviz, 59
	graphviz decode, 142
	graphviz encode, 142
	Has bundled edge with my_bundled_edge, 114
	Has bundled vertex with my_vertex, 80
	Has edge between vertices, 45
ed_, 25	header file, 12
edge, 45	
Edge descriptor, 24	idegree, 43
Edge descriptors, get, 25	in_degree, 43
Edge iterator, 23	Is isomorphic, 52
Edge iterator pair, 23	Is regular file, 146
Edge properties, 150	is_graphviz_friendly, 143
Edge, add, 21	link, 147
edges, 23, 25	Load directed bundled edges and vertices graph from dot, 122
Edges, counting, 16	Load directed bundled vertices graph from dot, 90
eip_, 23	Load directed custom edges and vertices graph from dot, 139
Empty directed graph, create, 12	Load directed graph from .dot, 60
Empty graph, create, 14	Load directed graph from dot, 61
	Load undirected bundled edges and vertices graph from dot, 126
Find first bundled edge with my_bundled_edge, 116	Load undirected bundled vertices graph from dot, 93
Find first bundled vertex with my_vertex, 82	Load undirected custom edges and vertices graph from dot, 141
Formed reference to void, 146	
get, 7	
Get bundled vertex my_bundled_vertex, 83	

Load undirected graph from .dot, 62	Set bundled edge my_bundled_edge, 119
Load undirected graph from _dot, 63	
Load undirected graph with bundled edges and vertices from .dot, 125	Set bundled vertex my_bundled_vertexes, 86
	Set graph name, 132
m_, 65, 96	Set vertex my_vertex, 84
make_bundled_vertices_writer, 87	static_assert, 17
member, 65, 96	static_cast, 15
my_bundled_edge, 96	std::copy, 21
my_bundled_edge declaration, 96	std::cout, 60
my_bundled_edge.h, 96	std::ifstream, 61
my_bundled_vertex, 65, 66	std::list, 13
my_bundled_vertex.h, 65	std::ofstream, 60
my_edge, 97	std::pair, 22
my_vertex declaration, 65	std::vector, 13
	STL, 13
No matching function for call to clear_out_edges, 147	test case, 6
node_id, 147	
noexcept, 12	Undefined reference to read_graphviz_new, 147
noexcept specification, 12	
Number of edges, get, 16	undirected graph, 9
Number of vertices, get, 15	unsigned long, 15
	vd, 22
out_degree, 43	vd_, 18
	Vertex descriptor, 18, 21
Path graph, create, 34	Vertex descriptors, get, 20
Petersen graph, create, 37	Vertex iterator, 19
Property not found, 147	Vertex iterator pair, 19
Property not found: node_id, 147, 148	Vertex iterators, get, 19
	Vertex properties, 150
read_graphviz_new, 147	Vertex, add, 17
read_graphviz_new, undefined reference, 147	vertices, 19, 21
	Vertices, counting, 15
S, 14, 129	vip_, 19
Save bundled edges and vertices graph to dot, 121	
Save bundled vertices graph to dot, 87	
Save graph as .dot, 59	
Save graph to dot, 60	
Save graph with bundled edges and vertices as .dot, 120	
Save graph with graph name to dot, 138	