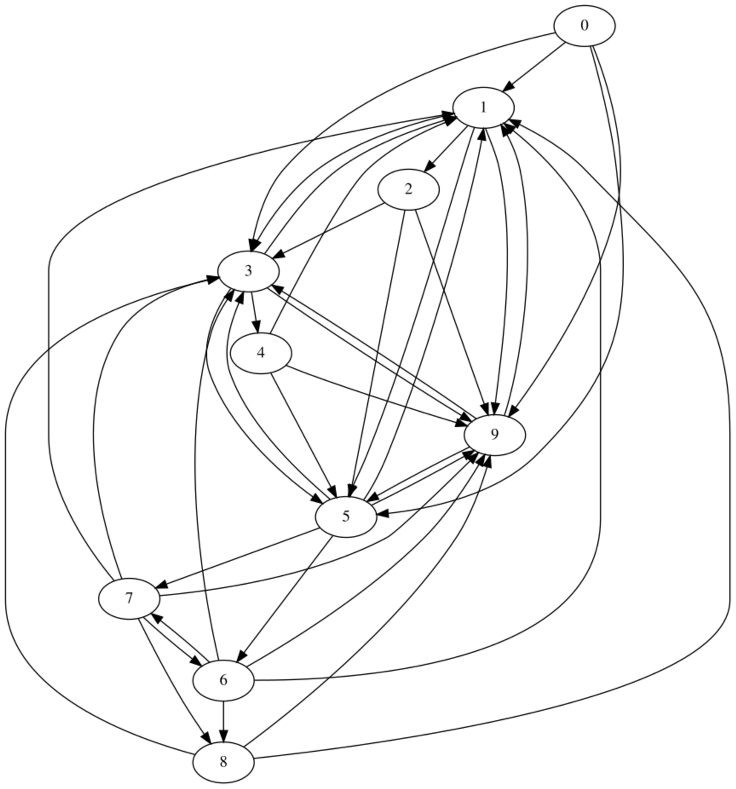
Boost.Graph Cookbook 1: Basics



Richèl J.C. Bilderbeek

January 3, 2022

Contents

Chapter 1

Introduction

This is 'Boost.Graph Cookbook 1: Basics', version 3.3.

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 Boost.Graph book [1] 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 that the book [1] 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 $\triangle$ 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 then 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 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 ([2], chapter 18.12.1 and [3], item 43). Sometimes using these algorithms becomes a burden on the lines of code. 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 GitHub Actions in both debug and release mode.

Tested to work All functions in this tutorial are tested, using the Boost.Test library. GitHub Actions 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`.

1.4 License

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

Figure 1.1: 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/danieljh`
- 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

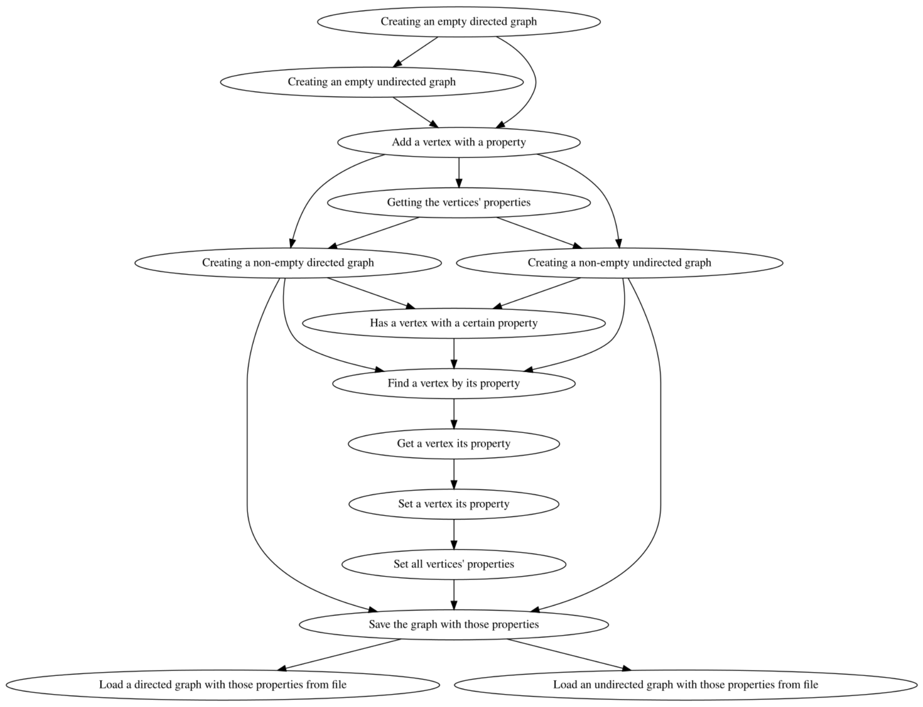

Creating an empty directed graph
Creating an empty undirected graph
Add a vertex with a property
Getting the vertices' properties
Creating a non-empty directed graph
Creating a non-empty undirected graph
Has a vertex with a certain property
Find a vertex by its property
Get a vertex its property
Set a vertex its property
Set all vertices' properties
Save the graph with those properties
Load a directed graph with those properties from file
Load an undirected graph with those properties from file

Figure 1.2: The relations between sub-chapters

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 1.2.

There are also some bonus chapters, that I have labeled with a $\triangle$. 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.

Chapter 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. a property can have than a vertex has a color or an edge that has a length).

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 2.1.

In a directed graph , an edge goes from a certain vertex, its source, to another (which may actually be the same), its target. A directed graph is shown in figure 2.2.

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: see chapter 2.14
- K_2, an undirected graph with two vertices and one edge, see chapter 2.19

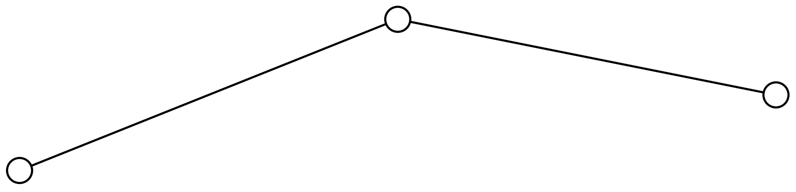


Figure 2.1: Example of an undirected graph

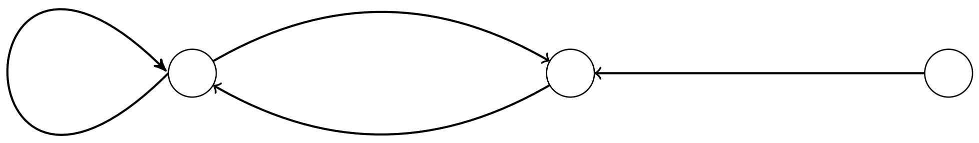


Figure 2.2: Example of a directed graph

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 2.9
- Edge descriptors: see chapter 2.12

After this chapter you may want to:

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

2.1 Creating an empty (directed) graph

Let's create an empty graph! Listing 2.1 shows the function to create an empty graph.

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

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

Listing 2.1: Creating an empty (directed) graph

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' [1].

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 [2], so it is preferred to mark it `noexcept` ([4], chapter 13.7)
- 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

Listing 2.2 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 [3]. 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, [4] chapter 31.6), which lets the compiler figure out the type itself.

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

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

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

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

[3] I do not think it is important to have creative names

Listing 2.2: Demonstration of create_empty_directed_graph

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 two different STL [4] containers. `std::vector` is the container you should use by default ([4], chapter 31.6, [5], 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 ([2], chapter 7.9.3, [4], chapter 12.7, [3], item 3, [6], chapter 3, [5], item 15, [7], FAQ 14.05, [8], item 8, [9], 9.1.6).

2.2 Creating an empty undirected graph

Let's create another empty graph! This time, we even make it undirected! Listing 2.3 shows how to create an undirected graph.

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

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

Listing 2.3: Creating an empty undirected graph

This algorithm differs from the `create_empty_directed_graph` function (algorithm 2.1) in that there are three template arguments that need to be specified in the creation of the `boost::adjacency_list`:

[4]Standard Template Library, the standard library

- 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

Listing 2.4 demonstrates the `create_empty_undirected_graph` function.

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

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

Listing 2.4: Demonstration of create_empty_undirected_graph

Congratulations, with algorithm 2.4, you have 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!

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

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;
}
```

Listing 2.5: Count the number of vertices

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 ([9], 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 ([2], chapter 4.4) should be avoided. The integer `n` is initialized using list-initialization, which is preferred over the other initialization syntaxes ([4], 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 ([2], chapter 24.5.18, [4], chapter 30.5, [5]. chapter 68, [10], chapter 8.2, [11], hour 24, [9], chapter 2.6).

The function `get_n_vertices` is demonstrated in algorithm 2.6, to measure the number of vertices of both the directed and undirected graph we are already able to create.

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

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);
}
```

Listing 2.6: Demonstration of the get_n_vertices function

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:

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

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;
}
```

Listing 2.7: Count the number of edges

This code is similar to the `get_n_vertices` function (algorithm 2.5, 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 2.8, to measure the number of edges of an empty directed and undirected graph.

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

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);
}
```

Listing 2.8: Demonstration of the get_n_edges function

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 2.9:

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

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;
}
```

Listing 2.9: Adding a vertex to a graph

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.

Listing 2.10 shows how to add a vertex to a directed and undirected graph.

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

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);
}
```

Listing 2.10: Demonstration of the add_vertex function

This demonstration code creates two empty graphs, adds one vertex to each and then `assert`s 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 a vertex, for example the vertex its out degrees (chapter 3.1)

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 2.11.

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`.

```
#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);
}
```

Listing 2.11: Get the vertex iterators of a graph

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

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

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);
}
```

Listing 2.12: Demonstration of get_vertex_iterators

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. Listing 2.13 shows how to obtain all vertex descriptors from a graph.

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

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;
}
```

Listing 2.13: Get all vertex descriptors of a graph

This is the first more complex piece of code. In the first lines, some `using` statements allow for shorter type names [5].

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 iterator over. std::copy is an STL algorithm to copy a half-open range. Prefer algorithm calls over hand-written for-loops ([2] chapter 18.12.1, [3] 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.

Listing 2.14 demonstrates that an empty graph has no vertex descriptors:

[5]which may be necessary just to create a tutorial with code snippets that are readable

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

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());
}
```

Listing 2.14: Demonstration of get_vertex_descriptors

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). Listing 2.15 adds two vertices to a graph, and connects these two using `boost::add_edge` :

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

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;
}
```

Listing 2.15: Adding (two vertices and) an edge to a graph

Listing 2.15

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 2.16, in which an edge is added to both a directed and undirected graph, after which the number of edges and vertices is checked.

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

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);
}
```

Listing 2.16: Demonstration of add_edge

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`. Listing 2.17 shows how to obtain these:

```
#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);
}
```

Listing 2.17: Get the edge iterators of a graph

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.

Listing 2.18 demonstrates `get_edge_iterators` by showing that both iterators of the edge iterator pair point to the same location, when the graph is empty.

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

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);
}
```

Listing 2.18: Demonstration of get_edge_iterators

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 named 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 2.13), as shown in algorithm 2.19:

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

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;
}
```

Listing 2.19: Get all edge descriptors of a graph

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

Listing 2.20 demonstrates the `get_edge_descriptor`, by showing that empty graphs do not have any edge descriptors.

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

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());
```

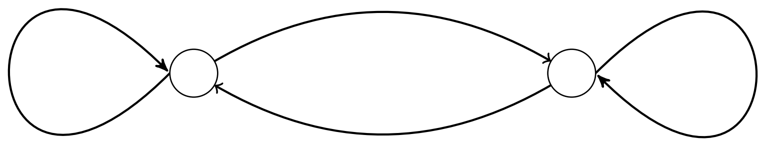


Figure 2.3: The two-state Markov chain

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

Listing 2.20: Demonstration of get_edge_descriptors

2.14 Creating a directed graph

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

This directed graph is a two-state Markov chain, with two vertices and four edges, as depicted in figure 2.3:

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

2.15 Function to create such a graph

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

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

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;
}
```

Listing 2.21: Creating the two-state Markov chain as depicted in figure 2.3

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.

2.16 Creating such a graph

Listing 2.22 demonstrates the `create_markov_chain_graph` function and checks if it has the correct amount of edges and vertices:

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

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);
}
```

Listing 2.22: Demonstration of the create_markov_chain

2.17 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 3.20). The .dot file created is displayed in algorithm 2.23:

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

Listing 2.23: .dot file created from the create_markov_chain_graph function (algorithm 2.21), converted from graph to .dot file using algorithm

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 -)

2.18 The .svg file produced

The .svg file of this graph is shown in figure 2.4:

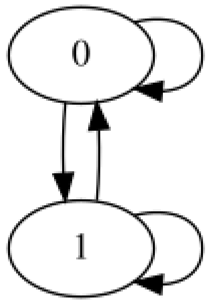


Figure 2.4: .svg file created from the create_markov_chain function (algorithm 2.21) its .dot file and converted from .dot file to .svg using algorithm 11.2;

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

2.19 Creating K_2, a fully connected undirected graph with two vertices

Finally, we are going to create an undirected non-empty 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 2.5.

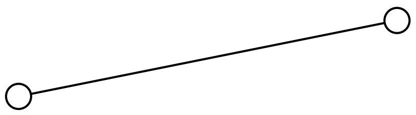


Figure 2.5: K_2 : a fully connected undirected graph with two vertices

2.20 Function to create such a graph

To create K_2, the following code can be used:

```
#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;
}
```

Listing 2.24: Creating K_2 as depicted in figure 2.5

This code is very similar to the `add_edge` function (algorithm 2.15). 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.

2.21 Creating such a graph

Listing 2.25 demonstrates how to `create_k2_graph` and checks if it has the correct amount of edges and vertices:

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

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);
}
```

Listing 2.25: Demonstration of create_k2_graph

2.22 The .dot file produced

Running a bit ahead, this graph can be converted to the .dot file as shown in algorithm 2.26:

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

Listing 2.26: .dot file created from the create_k2_graph function (algorithm 2.24) converted from graph to .dot file using algorithm 3.20

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 <>)

2.23 The .svg file produced

Continuing to running a bit ahead, this .dot file can be converted to the .svg as shown in figure 2.6:

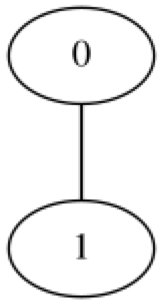


Figure 2.6: .svg file created from the create_k2_graph' function (algorithm 2.24) its .dot file, converted from .dot file to .svg using algorithm 11.2

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

2.24 △ Creating K_3, a fully connected undirected graph with three vertices

This is an extension of the previous chapter

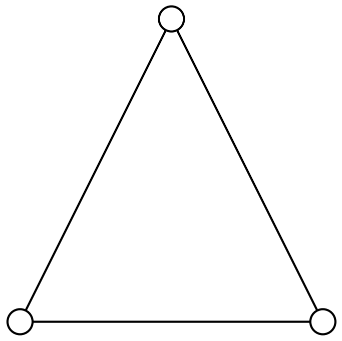


Figure 2.7: K_3: a fully connected graph with three edges and vertices

2.25 Graph

To create a fully connected undirected graph with three vertices (also called K_4), one needs three vertices and three (undirected) edge, as depicted in figure 2.7.

2.26 Function to create such a graph

To create K_3, the following code can be used:

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

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;
}
```

Listing 2.27: Creating K_3 as depicted in figure 2.7

2.27 Creating such a graph

Listing 2.28 demonstrates `create_k3_graph` and checks if it has the correct amount of edges and vertices:

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

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);
}
```

Listing 2.28: Demonstration of create_k3_graph

2.28 The .dot file produced

This graph can be converted to the .dot file as shown in algorithm 2.29:

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

Listing 2.29: .dot file created from the create_k3_graph function (algorithm 2.27) converted from graph to .dot file using algorithm 3.20

2.29 The .svg file produced

Continuing to running a bit ahead, this .dot file can be converted to the .svg as shown in figure 2.8:

2.30 △ Creating a path graph

A path graph is a linear graph without any branches

2.31 Graph

Here I show a path graph with four vertices (see figure 2.9):

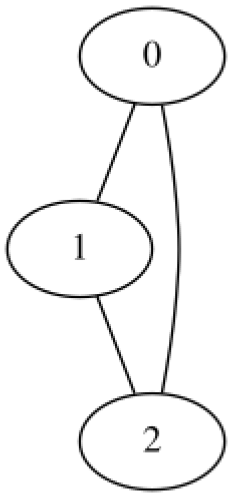


Figure 2.8: .svg file created from the create_k3_graph function (algorithm 2.27) its .dot file, converted from .dot file to .svg using algorithm 11.2

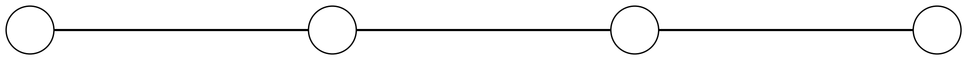


Figure 2.9: A path graph with four vertices

2.32 Function to create such a graph

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

```
#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;
}
```

Listing 2.30: Creating a path graph as depicted in figure 2.9

2.33 Creating such a graph

Listing 2.31 demonstrates `create_path_graph` and checks if it has the correct amount of edges and vertices:

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

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);
}
```

Listing 2.31: Demonstration of create_path_graph

2.34 The .dot file produced

This graph can be converted to the .dot file as shown in algorithm 2.32:

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

Listing 2.32: .dot file created from the create_path_graph function (algorithm 2.30) converted from graph to .dot file using algorithm 3.20

2.35 The .svg file produced

The .dot file can be converted to the .svg as shown in figure 2.10:

2.36 △ Creating a Peterson graph

A Petersen graph is the first graph with interesting properties.

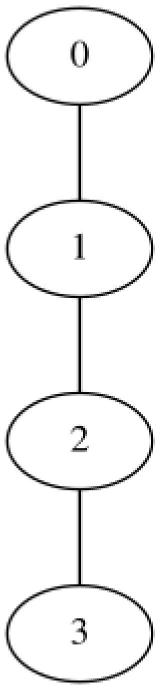


Figure 2.10: .svg file created from the create_path_graph function (algorithm 2.30) its .dot file, converted from .dot file to .svg using algorithm 11.2

2.37 Graph

To create a Petersen graph, one needs five vertices and five undirected edges, as depicted in figure 2.11.

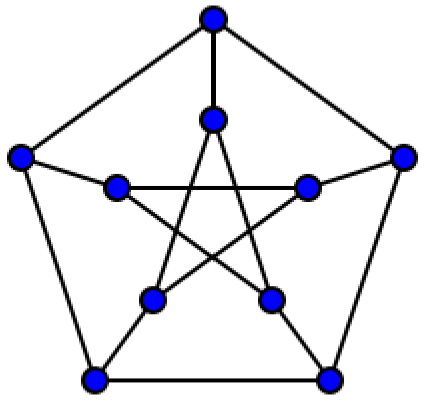


Figure 2.11: A Petersen graph (from https://en.wikipedia.org/wiki/Petersen_graph)

2.38 Function to create such a graph

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

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

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;
}
```

Listing 2.33: Creating Petersen graph as depicted in figure 2.11

2.39 Creating such a graph

Listing 2.34 demonstrates how to use `create_petersen_graph` and checks if it has the correct amount of edges and vertices:

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

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);
}
```

Listing 2.34: Demonstration of create_k3_graph

2.40 The .dot file produced

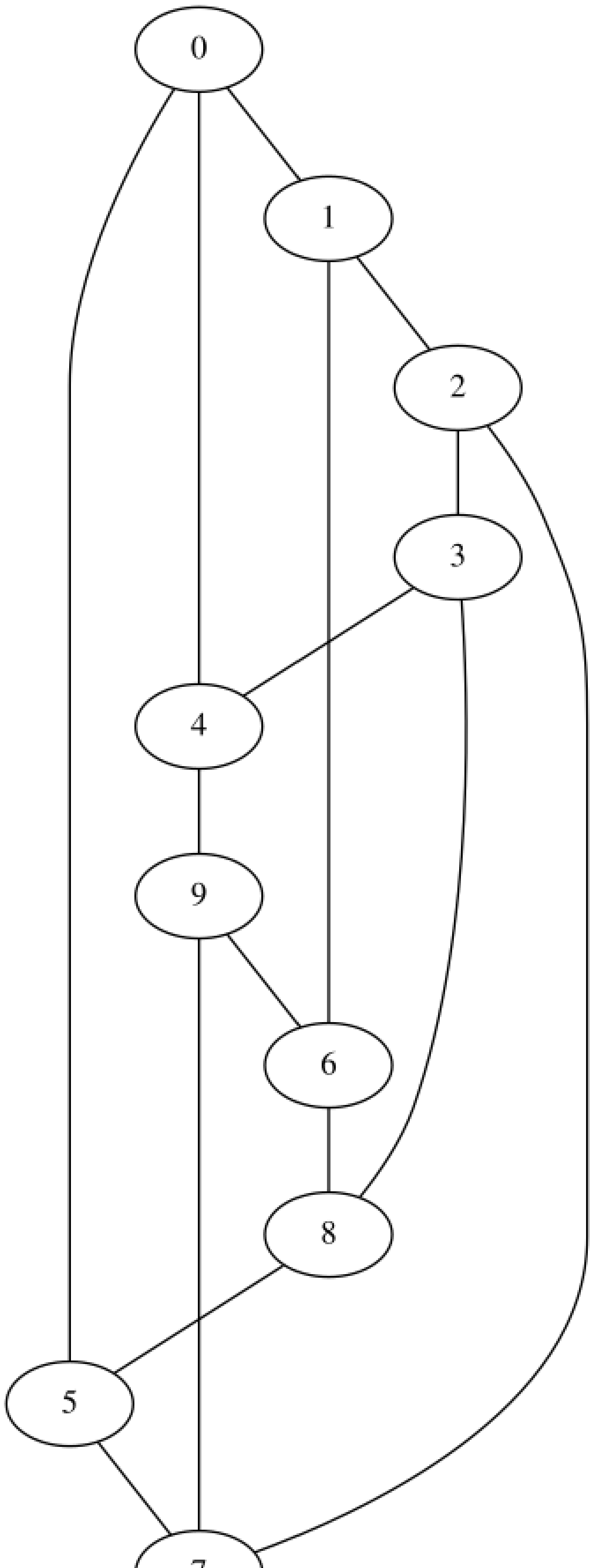
This graph can be converted to the .dot file as shown in algorithm 2.35:

```
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 ;
}
```

Listing 2.35: .dot file created from the create_petersen_graph function (algorithm 2.33) converted from graph to .dot file using algorithm 3.20

2.41 The .svg file produced

This .dot file can be converted to the .svg as shown in figure 2.12:


0
1
2
3
4
9
6
8
5
7

Chapter 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 3.10
- Loading an undirected graph without properties from .dot file: see chapter 3.12
- Loading a directed graph without properties from .dot file: see chapter 3.11

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`)

Listing 3.1 shows how to obtain these:

```
#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;
}
```

Listing 3.1: Get the vertices' out degrees

The structure of this algorithm is similar to `get_vertex_descriptors` (algorithm 2.13), 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 3.2.

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

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
      );
}
```

Listing 3.2: Demonstration of the get_vertex_out_degrees function

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:

```
#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;
}
```

Listing 3.3: Check if there exists an edge between two vertices

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).

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

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));
}
```

Listing 3.4: Demonstration of the has_edge_between_vertices function

3.3 △ Get the edge between two vertices

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

```
#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;
}
```

Listing 3.5: Get the edge between two vertices

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.

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

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);
}
```

Listing 3.6: Demonstration of the get_edge_between_vertices function

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:

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

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;
}
```

Listing 3.7: Get the direct-neighbour subgraph from a vertex descriptor

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

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

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);
  }
}
```

Listing 3.8: Demo of the create_direct_neighbour_subgraph function

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.

```
#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
     ());
}
```

Listing 3.9: Get the direct-neighbour subgraph from a vertex descriptor

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:

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

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;
}
```

Listing 3.10: Create all direct-neighbour subgraphs from a graph without properties

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

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

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);
  }
}
```

Listing 3.11: Demo of the create_all_direct_neighbour_subgraphs function

3.6.1 △ Are two graphs isomorphic?

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

```
#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);
}
```

Listing 3.12: Check if two graphs are isomorphic

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

```
#include "create_k3_graph.h"
#include "create_path_graph.h"
#include "is_isomorphic.h"
```

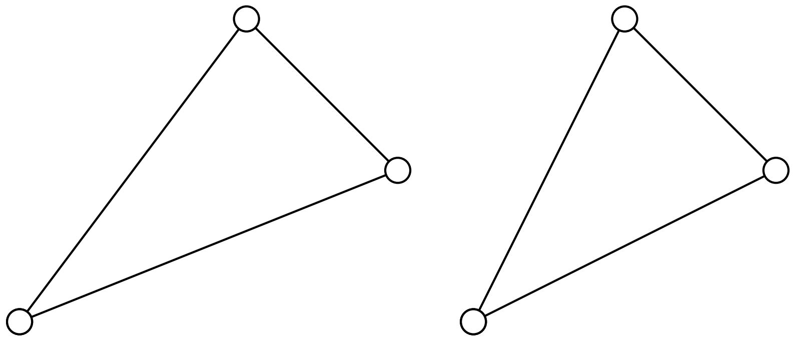


Figure 3.1: Example of a directed graph with two components

```
#include <boost/test/unit_test.hpp>

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));
}
```

Listing 3.13: Demo of the is_isomorphic function

3.7 △△ 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.

This algorithm counts the number of connected components:

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

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;
}
```

Listing 3.14: Count the number of connected components

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

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

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

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);
}
```

Listing 3.15: Demo of the count_directed_graph_connected_components function

3.8 △△ 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, as shown in figure 3.3:

This algorithm counts the number of connected components:

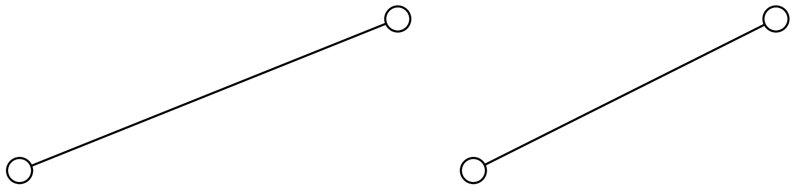


Figure 3.2: Example of an undirected graph with two components

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

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)));
}
```

Listing 3.16: Count the number of connected components

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

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

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

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);
}
```

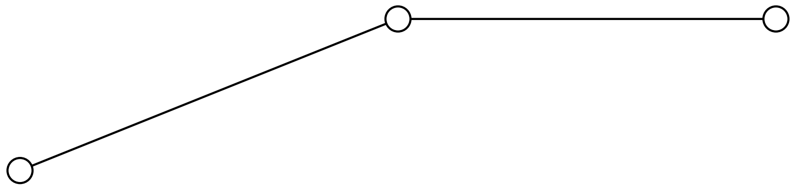


Figure 3.3: Example of an undirected graph with two levels

Listing 3.17: Demo of the count_undirected_graph_connected_components function

3.9 △△ 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, as shown in figure 3.3:

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.

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

// 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;
}
```

Listing 3.18: Count the number of levels in an undirected graph

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.

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

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);
}
```

Listing 3.19: Demo of the count_undirected_graph_levels function

3.10 Saving a graph to a .dot file

Graph are easily saved to a file, thanks to Graphviz. Graphviz (short for Graph Visualization Software) is a package of open-source tools for drawing graphs. It uses the DOT language for describing graphs, and these are commonly stored in (plain-text) .dot files (I show .dot file of every non-empty graph created, e.g. chapters 2.14 and 2.19)

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

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);
}
```

Listing 3.20: Saving a graph to a .dot file

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.

Listing 3.21 shows how to use the `save_graph_to_dot` function:

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

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");
}
```

Listing 3.21: Demonstration of the save_graph_to_dot function

When using the `save_graph_to_dot` function (algorithm 3.20), only the structure of the graph is saved: all other properties (e.g vertex names, edge lengths) are not stored.

3.11 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 3.22:

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

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);
  auto g = create_empty_directed_graph();
  boost::dynamic_properties dp(boost::
      ignore_other_properties);
  boost::read_graphviz(f, g, dp);
  return g;
}
```

Listing 3.22: Loading a directed graph from a .dot file

In this algorithm, first it is checked if the file to load exists, using the `is_regular_file` function (algorithm 11.3), 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.

Listing 3.23 shows how to use the `load_directed_graph_from_dot` function:

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

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));
}
```

Listing 3.23: Demonstration of the load_directed_graph_from_dot function

This demonstration shows how the Markov chain is created using the `create_markov_chain_graph` function (algorithm 2.21), saved and then loaded. The loaded graph is then checked to be a two-state Markov chain.

3.12 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 3.11. Listing 3.24 show how to do so:

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

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);
  auto g = create_empty_undirected_graph();
  boost::dynamic_properties p(boost::ignore_other_properties
      );
  boost::read_graphviz(f, g, p);
  return g;
}
```

Listing 3.24: Loading an undirected graph from a .dot file

The only difference with loading a directed graph, is that the initial empty graph is undirected instead.

Chapter 3.11 describes the rationale of this function.

Listing 3.25 shows how to use the `load_undirected_graph_from_dot` function:

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

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));
}
```

Listing 3.25: Demonstration of the load_undirected_graph_from_dot function

This demonstration shows how the K_2 graph is created using the `create_k2_graph` function (algorithm 2.24), saved and then loaded. The loaded graph is checked to be a K_2 graph.

Chapter 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 [1]. The following graphs will be created:

- An empty directed graph that allows for bundled vertices: see chapter 4.2
- 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
- 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:

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

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

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;
  const std::string& get_description() const noexcept;
  const std::string& get_name() const noexcept;
  double get_x() const noexcept;
  double get_y() const 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;
bool operator<(
  const my_bundled_vertex& lhs, const my_bundled_vertex& rhs
      ) noexcept;
```

Listing 4.1: Declaration of my_bundled_vertex

`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

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

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

Listing 4.2: Creating an empty directed graph with bundled vertices

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

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

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

Listing 4.3: Creating an empty undirected graph with bundled vertices

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

4.4 Add a bundled vertex

Adding a vertex without a name was trivially easy (see chapter 2.5). Adding a bundled vertex takes slightly more work, as shown by algorithm 4.4:

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

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);
}
```

Listing 4.4: Add a bundled vertex

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

[2]

When the vertices of a graph have any bundled `my_bundled_vertex`, one can extract these as such:

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

[2]the name my_vertexes; is chosen to allows you to replace my_vertex by your favorite datatype name, although in English the plural of vertex is vertices

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

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;
}
```

Listing 4.5: Get the bundled vertices' my_vertexes

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 4.1 shows the graph that will be reproduced:

4.6.2 Function to create such a graph

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

```
#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);
```

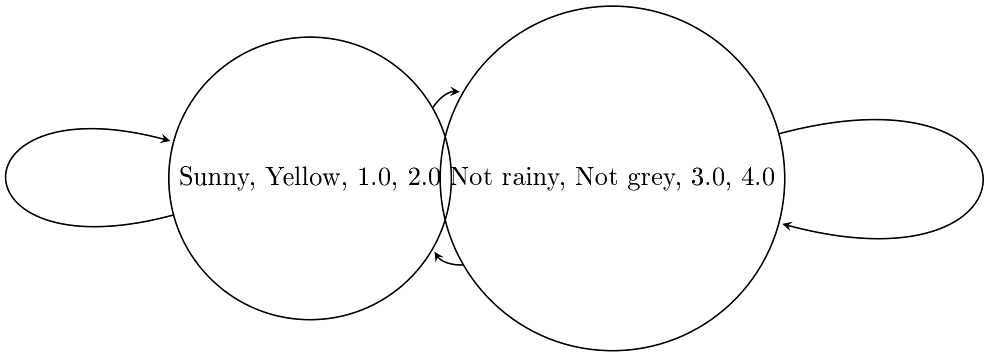


Figure 4.1: A two-state Markov chain where the vertices have bundled properties and the edges have no properties. The vertices' properties are nonsensical

```
  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;
}
```

Listing 4.6: Creating the two-state Markov chain as depicted in figure 4.1

4.6.3 Creating such a graph

Here is the demo:

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

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);
}
```

Listing 4.7: Demo of the create_bundled_vertices_markov_chain function (algorithm 4.6)

4.6.4 The .dot file produced

```
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 ;
}
```

Listing 4.8: .dot file created from the create_bundled_vertices_markov_chain function (algorithm 4.6) converted from graph to .dot file using algorithm 5.10

4.6.5 The .svg file produced

4.7 Creating K_2 with bundled vertices

4.7.1 Graph

We reproduce the K_2 without properties of chapter 2.19, but with our bundled vertices instead, as show in figure 4.3:

4.7.2 Function to create such a graph

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

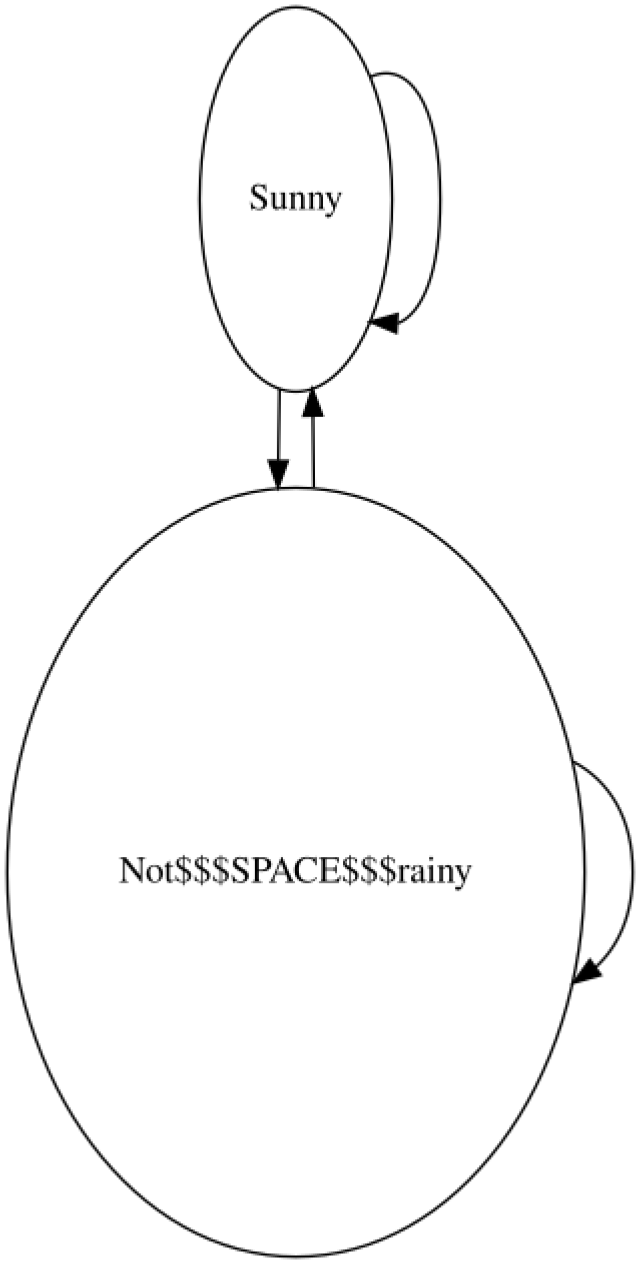


Figure 4.2: .svg file created from the create_bundled_vertices_markov_chain function (algorithm 4.6) its .dot file, converted from .dot file to .svg using algorithm 11.2

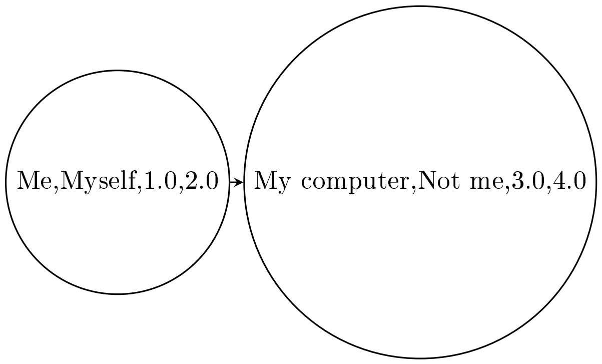


Figure 4.3: K_2: a fully connected graph with two bundled vertices

```
  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;
}
```

Listing 4.9: Creating K_2 as depicted in figure 4.3

Most of the code is a slight modification of the `create_k2_graph` function (algorithm 2.24). 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:

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

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)
       );
}
```

Listing 4.10: Demo of the create_bundled_vertices_k2_graph function (algorithm 4.9)

4.7.4 The .dot file produced

```
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 ;
}
```

Listing 4.11: .dot file created from the create_bundled_vertices_k2_graph function (algorithm 4.9) converted from graph to .dot file using algorithm 3.20

4.7.5 The .svg file produced

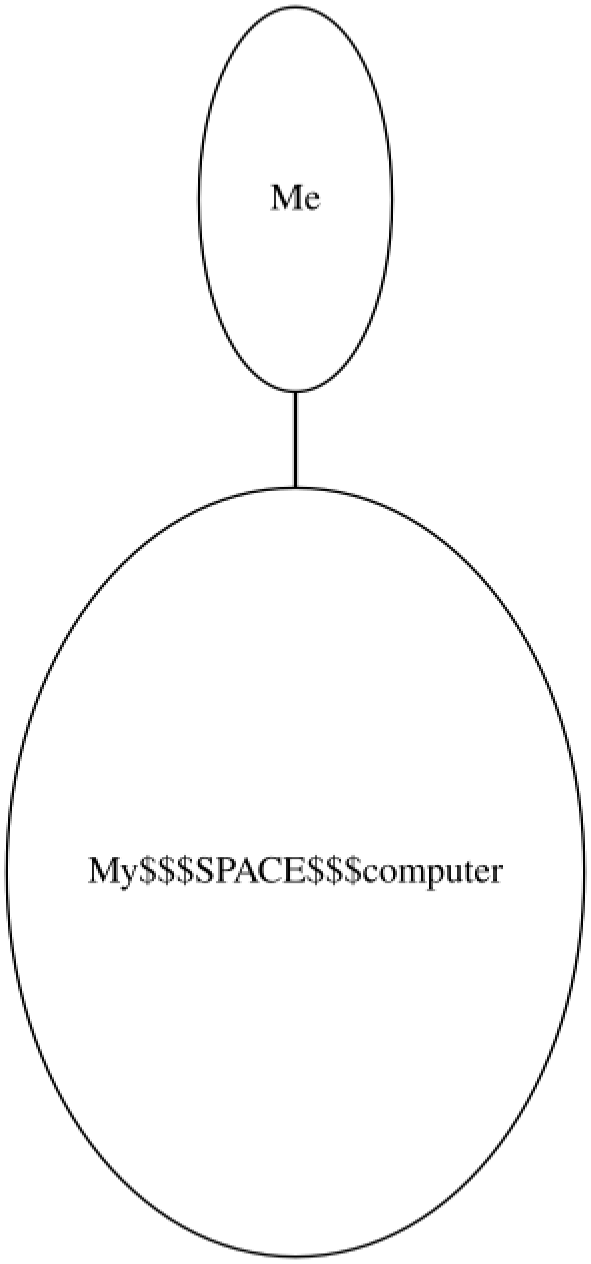


Figure 4.4: .svg file created from the create_bundled_vertices_k2_graph function (algorithm 4.9) its .dot file, converted from .dot file to .svg using algorithm 11.2

Chapter 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.

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

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;
}
```

Listing 5.1: Find if there is vertex with a certain my_bundled_vertex

This function can be demonstrated as in algorithm 5.2, where a certain my_bundled_vertex cannot be found in an empty graph. After adding the desired my_bundled_vertex, it is found.

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

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));
}
```

Listing 5.2: Demonstration of the has_bundled_vertex_with_my_vertex function

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. Listing 5.3 shows how to obtain a vertex descriptor to the first vertex found with a specific `my_bundled_vertex` value.

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

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( //Cannot use std::find
    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;
}
```

Listing 5.3: Find the first vertex with a certain my_bundled_vertex

With the vertex descriptor obtained, one can read and modify the vertex and the edges surrounding it. Listing 5.4 shows some examples of how to do so.

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

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);
}
```

Listing 5.4: Demonstration of the find_first_bundled_vertex_with_my_vertex function

5.3 Get a bundled vertex its my_bundled_vertex

To obtain the `my_bundled_vertex` from a vertex descriptor is simple:

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

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

Listing 5.5: Get a bundled vertex its my_vertex from its vertex descriptor

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. Listing 5.6 shows a simple example.

```
#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 <boost/test/unit_test.hpp>

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);
}
```

Listing 5.6: Demonstration if the get_bundled_vertex_my_vertex function

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 5.7

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

template <typename graph, typename my_bundled_vertex>
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;
}
```

Listing 5.7: Set a bundled vertex its my_vertex from its vertex descriptor

To use `set_bundled_vertex_my_vertex`, one first needs to obtain a vertex descriptor. Listing 5.8 shows a simple example.

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

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);
}
```

Listing 5.8: Demonstration if the set_bundled_vertex_my_vertex function

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:

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

template <typename graph, typename my_bundled_vertex>
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;
  }
}
```

Listing 5.9: Setting the bundled vertices' my_bundled_vertex-es

5.6 Storing a graph with bundled vertices as a .dot

If you used the `create_bundled_vertices_k2_graph` function (algorithm 4.9) to produce a K_2 graph with vertices associated with `my_bundled_vertex` objects, you can store these with algorithm 5.10:

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

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
      ));
}
```

Listing 5.10: Storing a graph with bundled vertices as a .dot file

This code looks small, because we call the `make_bundled_vertices_writer` function, which is shown in algorithm 5.11:

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

Listing 5.11: The make_bundled_vertices_writer function

Also this function is forwarding the real work to the `bundled_vertices_writer`, shown in algorithm 5.12:

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

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;
};
```

Listing 5.12: The bundled_vertices_writer function

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 A.2 for most Graphviz attributes) as shown in table 5.1:

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 5.1: 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 5.13:

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

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);
  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);
  }

  return g;
}
```

Listing 5.13: Loading a directed graph with bundled vertices from a .dot file

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.

Listing 5.14 shows how to use the `load_directed_bundled_vertices_graph_from_dot` function:

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

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));
}
```

Listing 5.14: Demonstration of the load_directed_bundled_vertices_graph_from_dot function

This demonstration shows how the Markov chain is created using the `create_bundled_vertices_mark` function (algorithm 4.6), 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 5.15:

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

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);
  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;
}
```

Listing 5.15: Loading an undirected graph with bundled vertices from a .dot file

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.

Listing 5.16 shows how to use the `load_undirected_bundled_vertices_graph_from_dot` function:

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

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));
}
```

Listing 5.16: Demonstration of the load_undirected_bundled_vertices_graph_from_dot function

This demonstration shows how K_2 with bundled vertices is created using the `create_bundled_vertices_k2_graph` function (algorithm 4.9), saved and then loaded. The loaded graph is checked to be the same as the original.

Chapter 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 [1].

- 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.

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

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

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;
  const std::string& get_description() const noexcept;
  const std::string& get_name() const noexcept;
  double get_height() const noexcept;
  double get_width() const 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;
bool operator<(const my_bundled_edge& lhs, const
    my_bundled_edge& rhs) noexcept;
```

Listing 6.1: Declaration of my_bundled_edge

`my_bundled_edge` is a class that has multiple properties: two doubles `m_width` (`m_` stands for member) and `m_height`, and two std::strings `m_name` and `m_description`. `my_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

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

```
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 {};
}
```

Listing 6.2: Creating an empty directed graph with bundled edges and vertices

This code is very similar to the code described in chapter 4.2, 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:

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

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);
}
```

Listing 6.3: Demonstration of the create_empty_directed_bundled_edges_and_vertices_graph function

6.3 Create an empty undirected graph with bundled edges and vertices

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

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 {};
}
```

Listing 6.4: Creating an empty undirected graph with bundled edges and vertices

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:

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

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);
}
```

Listing 6.5: Demonstration of the create_empty_undirected_bundled_edges_and_vertices_graph function

6.4 Add a bundled edge

Adding a bundled edge is very similar to adding an edge without properties (chapter 2.9).

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

```
#include <cassert>
#include <sstream>
#include <stdexcept>

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;
}
```

Listing 6.6: Add a bundled edge

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:

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

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);
}
```

Listing 6.7: Demo of add_bundled_edge

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:

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

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;
}
```

Listing 6.8: Get the edges' my_bundled_edges

The `my_bundled_edge` object associated with the edges are obtained from the graph its `property_map\verb` 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 6.1 shows the graph that will be reproduced:

6.6.2 Function to create such a graph

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

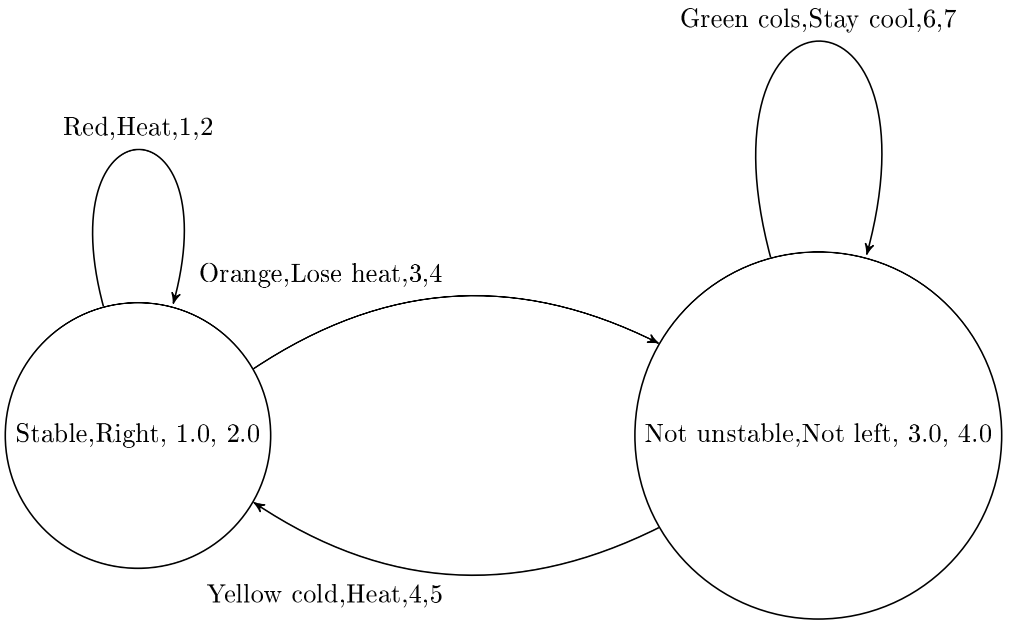


Figure 6.1: A two-state Markov chain where the edges and vertices have bundled properties. The edges' and vertices' properties are nonsensical

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

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;
}
```

Listing 6.9: Creating the two-state Markov chain as depicted in figure 6.1

6.6.3 Creating such a graph

Here is the demo:

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

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);
}
```

Listing 6.10: Demo of the create_bundled_edges_and_vertices_markov_chain function (algorithm 6.9)

6.6.4 The .dot file produced

```
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];
}
```

Listing 6.11: .dot file created from the create_bundled_edges_and_vertices_markov_chain function (algorithm 6.9) converted from graph to .dot file using algorithm 3.20

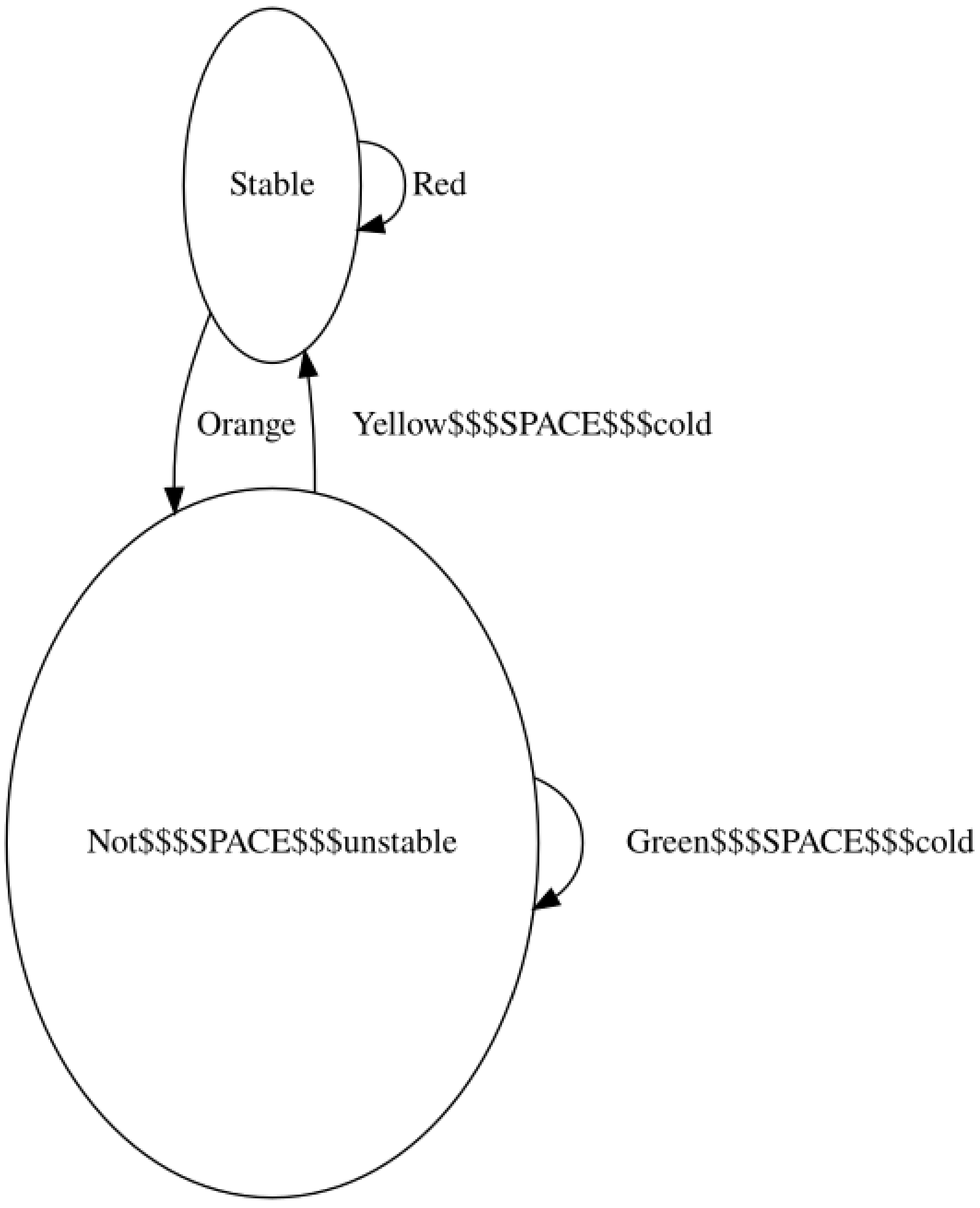


Figure 6.2: .svg file created from the create_bundled_edges_and_vertices_markov_chain function (algorithm 4.6) its .dot file converted from .dot file to .svg using algorithm 11.2

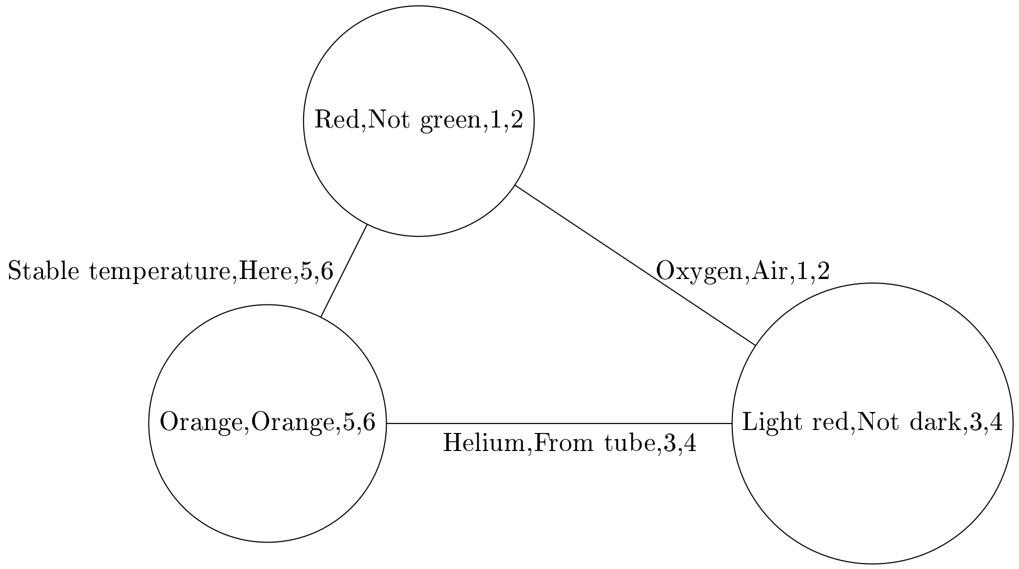


Figure 6.3: K_3: a fully connected graph with three bundled edges and vertices

6.6.5 The .svg file produced

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 without properties of chapter 2.24, but with our bundled edges and vertices instead:

6.7.2 Function to create such a graph

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

Listing 6.12: Creating K_3 as depicted in figure 6.3

Most of the code is a slight modification of algorithm 2.27. 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:

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

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);
}
```

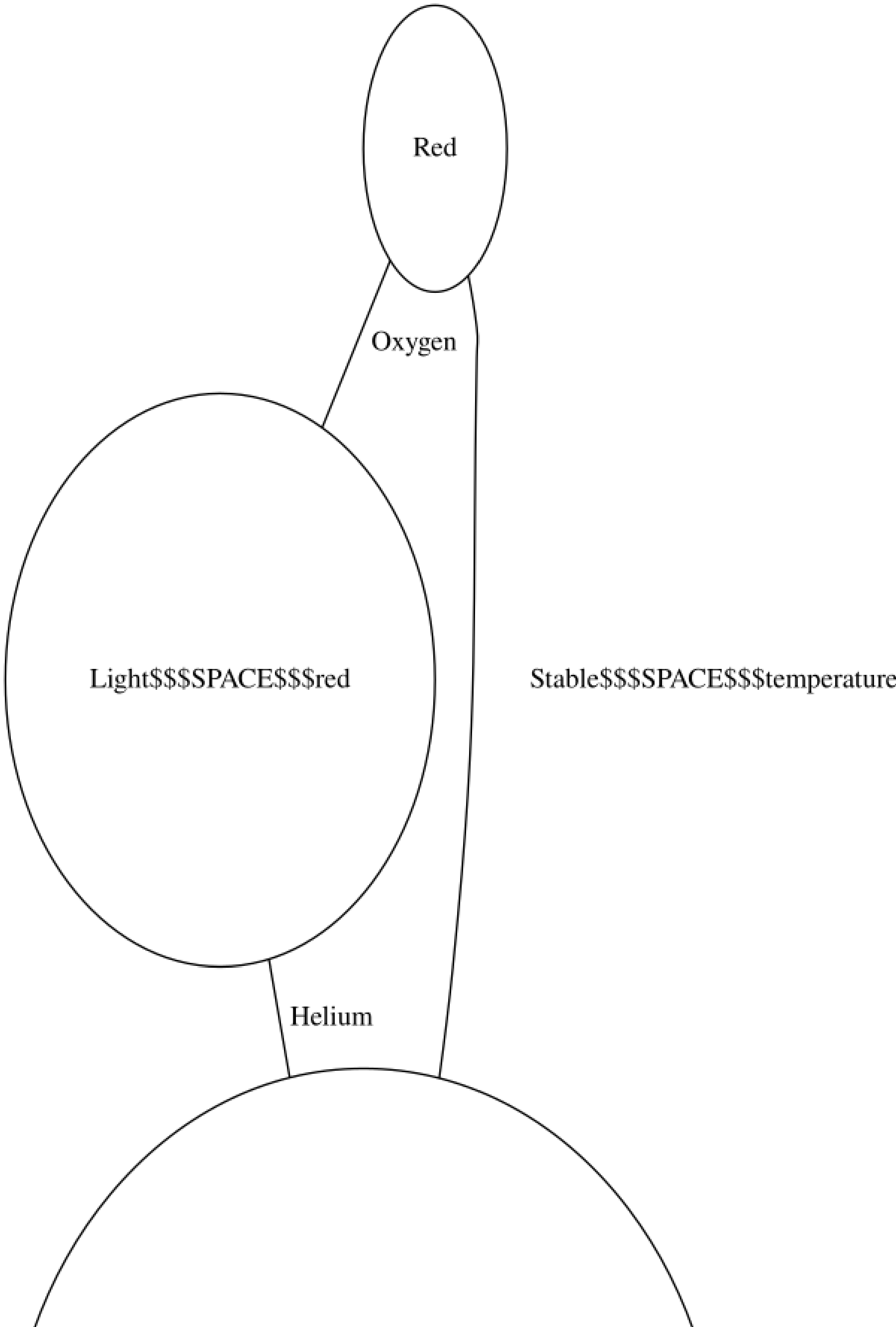
Listing 6.13: Demo of the create_bundled_edges_and_vertices_k3_graph function (algorithm 6.12)

6.7.4 The .dot file produced

```
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];
}
```

Listing 6.14: .dot file created from the create_bundled_edges_and_vertices_markov_chain function (algorithm 6.12) converted from graph to .dot file using algorithm 3.20

6.7.5 The .svg file produced


Red
Oxygen
Light$$$SPACE$$$red
Stable$$$SPACE$$$temperature
Helium

Chapter 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.

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

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;
}
```

Listing 7.1: Find if there is a bundled edge with a certain my_bundled_edge

This function can be demonstrated as in algorithm 7.2, where a certain `my_bundled_edge` cannot be found in an empty graph. After adding the desired `my_bundled_edge`, it is found.

```
#include "create_bundled_edges_and_vertices_k3_graph.h"
#include "has_bundled_edge_with_my_edge.h"
```

```
#include <boost/test/unit_test.hpp>

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));
}
```

Listing 7.2: Demonstration of the has_bundled_edge_with_my_edge function

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. Listing 7.3 shows how to obtain an edge descriptor to the first edge found with a specific `my_bundled_edge` value.

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

template <typename graph, typename my_bundled_edge>
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;
}
```

Listing 7.3: Find the first bundled edge with a certain my_bundled_edge

With the edge descriptor obtained, one can read and modify the edge and the vertices surrounding it. Listing 7.4 shows some examples of how to do so.

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

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));
}
```

Listing 7.4: Demonstration of the find_first_bundled_edge_with_my_edge function

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.

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

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

Listing 7.5: Get a vertex its my_bundled_vertex from its vertex descriptor

To use `get_my_bundled_edge`, one first needs to obtain an edge descriptor. Listing 7.6 shows a simple example.

```
#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 <boost/test/unit_test.hpp>
```

```

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);
}
```

Listing 7.6: Demonstration if the get_my_bundled_edge function

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 7.7.

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

template <typename graph, typename my_bundled_edge>
void set_my_bundled_edge(const my_bundled_edge& edge,
  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] = edge;
}
```

Listing 7.7: Set a bundled edge its my_bundled_edge from its edge descriptor

To use `set_bundled_edge_my_edge`, one first needs to obtain an edge descriptor. Listing 7.8 shows a simple example.

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

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);
}
```

Listing 7.8: Demonstration if the set_bundled_edge_my_edge function

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 6.12) 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 7.9:

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

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));
}
```

Listing 7.9: Storing a graph with bundled edges and vertices as a .dot file

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 7.10:

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

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);
  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
  {
    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);
    }
  }

  // Decode edges
  {
    const auto eip = edges(g);
    const auto j = eip.second;
#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wmaybe-uninitialized"
    for (auto i = eip.first; i != j; ++i) {
      g[*i].m_name = graphviz_decode(g[*i].m_name);
      g[*i].m_description = graphviz_decode(g[*i].
          m_description);
    }
#pragma GCC diagnostic pop
  }

  return g;
}
```

Listing 7.10: Loading a directed graph with bundled edges and vertices from a .dot file

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.

Listing 7.11 shows how to use the `load_directed_bundled_edges_and_vertices_graph_from_dot` function:

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

```
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));
}
```

Listing 7.11: Demonstration of the load_directed_bundled_edges_and_vertices_graph_from_dot function

This demonstration shows how the Markov chain is created using the `create_bundled_edges_and_ver` function (algorithm 6.9), 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 7.12:

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

template <class graph>
graph
    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);
  graph g;

  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
  {
    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);
    }
  }

  // Decode edges
  {
    const auto eip = edges(g);
    const auto j = eip.second;
    for (auto i = eip.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;
}
```

Listing 7.12: Loading an undirected graph with bundled edges and vertices from a .dot file

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.

Listing 7.13 shows how to use the `load_undirected_bundled_vertices_graph_from_dot` function:

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

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
        <decltype(
      create_bundled_edges_and_vertices_k3_graph())>(
          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));
}
```

Listing 7.13: Demonstration of the load_undirected_bundled_edges_and_vertices_graph_from_dot function

This demonstration shows how K_2 with bundled vertices is created using the `create_bundled_vertices_k2_graph` function (algorithm 4.9), saved and then loaded. The loaded graph is checked to be a graph similar to the original.

Chapter 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

Listing 8.1 shows the function to create 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 {};
}
```

Listing 8.1: Creating an empty directed graph with a graph name

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

Listing 8.2 demonstrates the `create_empty_directed_graph_with_graph_name` function.

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

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);
}
```

Listing 8.2: Demonstration of create_empty_directed_graph_with_graph_name

8.2 Create an empty undirected graph with a graph name property

Listing 8.3 shows the function to create 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 {};
}
```

Listing 8.3: Creating an empty undirected graph with a graph name

This code is very similar to the code described in chapter 8.1, except that the directness (the third template argument) is undirected (due to the boost::undirectedS).

Listing 8.4 demonstrates the `create_empty_undirected_graph_with_graph_name` function.

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

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);
}
```

Listing 8.4: Demonstration of create_empty_undirected_graph_with_graph_name

8.3 Get a graph its name property

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

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

Listing 8.5: Get a graph its name

Listing 8.6 demonstrates the `get_graph_name` function.

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

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);
}
```

Listing 8.6: Demonstration of get_graph_name

8.4 Set a graph its name property

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

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;
}
```

Listing 8.7: Set a graph its name

Listing 8.8 demonstrates the `set_graph_name` function.

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

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);
}
```

Listing 8.8: Demonstration of set_graph_name

8.5 Create a directed graph with a graph name property

8.5.1 Graph

See figure 2.3.

8.5.2 Function to create such a graph

Listing 8.9 shows the function to create an empty directed graph with a graph name.

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

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;
}
```

Listing 8.9: Creating a two-state Markov chain with a graph name

8.5.3 Creating such a graph

Listing 8.10 demonstrates the `create_markov_chain_with_graph_name` function.

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

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")
     ;
}
```

Listing 8.10: Demonstration of create_markov_chain_with_graph_name

8.5.4 The .dot file produced

8.5.5 The .svg file produced

8.6 Create an undirected graph with a graph name property

8.6.1 Graph

See figure 2.5.

8.6.2 Function to create such a graph

Listing 8.11 shows the function to create 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;
}
```

Listing 8.11: Creating a K2 graph with a graph name

8.6.3 Creating such a graph

Listing 8.12 demonstrates the `create_k2_graph_with_graph_name` function.

```
#include "create_k2_graph_with_graph_name.h"
#include "get_graph_name.h"
```

```
#include <boost/test/unit_test.hpp>

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");
}
```

Listing 8.12: Demonstration of create_k2_graph_with_graph_name

8.6.4 The .dot file produced

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

Listing 8.13: .dot file created from the create_k2_graph_with_graph_name function (algorithm 8.11) converted from graph to .dot file using algorithm 3.20

8.6.5 The .svg file produced

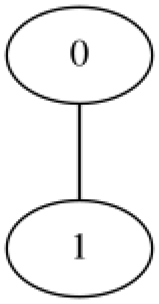


Figure 8.1: .svg file created from the create_k2_graph_with_graph_name function (algorithm 8.11) its .dot file, converted from .dot file to .svg using algorithm 11.2

Chapter 9

Working on graphs with a graph name

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

This works:

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

#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"; });
}
```

Listing 9.1: Storing a graph with a graph name as a .dot file

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:

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

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);
  boost::read_graphviz(f, g, dp);
  return g;
}
```

Listing 9.2: Loading a directed graph with a graph name from a .dot file

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:

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

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);
  boost::read_graphviz(f, g, dp);
  return g;
}
```

Listing 9.3: Loading an undirected graph with a graph name from a .dot file

Chapter 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.

```
#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;
}
```

Listing 10.1: Encode a std::string to a Graphviz-friendly format

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

This function undoes the `graphviz_encode` function (algorithm 10.1) and thus converts a Graphviz-friendly std::string to the original human-friendly std::string.

```
#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;
}
```

Listing 10.2: Decode a std::string from a Graphviz-friendly format to a human-friendly format

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.

```
#include "graphviz_encode.h"

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

Listing 10.3: Check if a std::string is Graphviz-friendly

Chapter 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`).

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

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;
}
```

Listing 11.1: Getting a data type its name as a std::string

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.

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

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 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());
  }
}
```

Listing 11.2: Convert a .dot file to a .svg

`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.

```
#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();
}
```

Listing 11.3: Check if a file exists

Chapter 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 12.1:

```
#include "create_k2_graph.h"
#include "get_vertex_names.h"

void formed_reference_to_void() noexcept
{
  get_vertex_names(create_k2_graph());
}
```

Listing 12.1: Creating the error 'formed reference to void'

In algorithm 12.1 a graph is created with vertices of no properties. Then the names of these vertices, which do not exists, are tried to be read. If you want to read the names of the vertices, supply a graph that has this property.

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.

```
#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);
}
```

Listing 12.2: Creating the error 'no matching function for call to clear_out_edges'

In algorithm 12.2 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_

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 3.12) 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. Listing 12.3 shows how to trigger this run-time error.

```
#include "create_empty_undirected_graph.h"
#include "create_k2_graph.h"
#include "is_regular_file.h"
#include "save_graph_to_dot.h"
#include <boost/graph/graphviz.hpp>
#include <cassert>
#include <fstream>

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);
  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");
}
```

Listing 12.3: Creating the error 'Property not found: node_id'

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);
  graph g;
  boost::dynamic_properties dp;
  dp.property(TODO}node_id}, get(boost::vertex_custom_type, g));
  dp.property(TODO}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);
  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;
}
```

Bibliography

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

[2] Bjarne Stroustrup. *The C++ Programming Language (3rd edition)*. 1997. ISBN 0-201-88954-4.

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

[4] Bjarne Stroustrup. *The C++ Programming Language (4th edition)*. 2013. ISBN 978-0-321-56384-2.

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

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

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

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

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

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

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

Appendix A

Appendix

A.1 List of all edge, graph and vertex properties

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

A.2 Graphviz attributes

List created from `www.graphviz.org/content/attrs`, where only the attributes that are supported by all formats are listed:

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

Edge	Graph	Vertex
arrowhead	_background	color
arrowsize	bgcolor	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	

Index