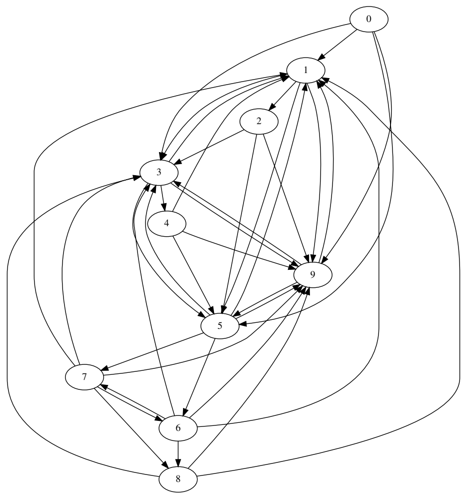
Boost.Graph Cookbook 2: Algorithms

John Zhang Zongren

May 11, 2018



Contents

1 Introduction

This is 'Boost.Graph Cookbook 2: Algorithms' version 0.1.

1.1 Why this tutorial

[Text from John here]

1.2 Tutorial style

Readable for beginners This tutorial is aimed at the beginner programmer. This tutorial is intended to take the reader to the level of understanding the book [2] and the Boost.Graph website require. It is about basic graph manipulation, not the more advanced graph algorithms.

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

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

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

1.3 Coding style

Concept For every concept, I will show

- a function that achieves a goal, for example 'create_empty_undirected_graph'
- a test case of that function, that demonstrates how to use the function, for example 'create_empty_undirected_graph_test'

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

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 ([3] chapter 18.12.1, [1] item 43). Sometimes using these algorithms becomes a burden on the lines of code. This is because in C++11, a lambda function argument (use by the algorithm) must have its data type specified. It may take multiple lines of ‘using’ statements being able to do so. In C++14 one can use ‘auto’ there as well. So, only if it shortens the number of lines significantly, I use raw for-loops, even though you shouldn’t.

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

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

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

Availability The code, as well as this tutorial, can be downloaded from the GitHub at `www.github.com/richelbilderbeek/boost_graph_cookbook_2`.

1.4 License

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

Figure 1: Creative Commons license 4.0

1.5 Feedback

This tutorial is not intended to be perfect yet. For that, I need help and feedback from the community. All referenced feedback is welcome, as well as any constructive feedback.

I have tried hard to strictly follow the style as described above. If you find I deviated from these decisions somewhere, I would be grateful if you'd let know. Next to this, there are some sections that need to be coded or have its code improved.

1.6 Acknowledgements

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

- Richel Bilderbeek

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 3

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

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

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

2 Graphs without properties

Boost.Graph is about creating graphs. In this chapter we create the simplest of graphs, in which edges and nodes have no properties (e.g. having a name).

Still, there are two types of graphs that can be constructed: undirected and directed graphs. The difference between directed and undirected graphs is in the edges: in an undirected graph, an edge connects two vertices without any directionality, as displayed in figure 3. In a directed graph, an edge goes from a certain vertex, its source, to another (which may actually be the same), its target. A directed graph is shown in figure 4.

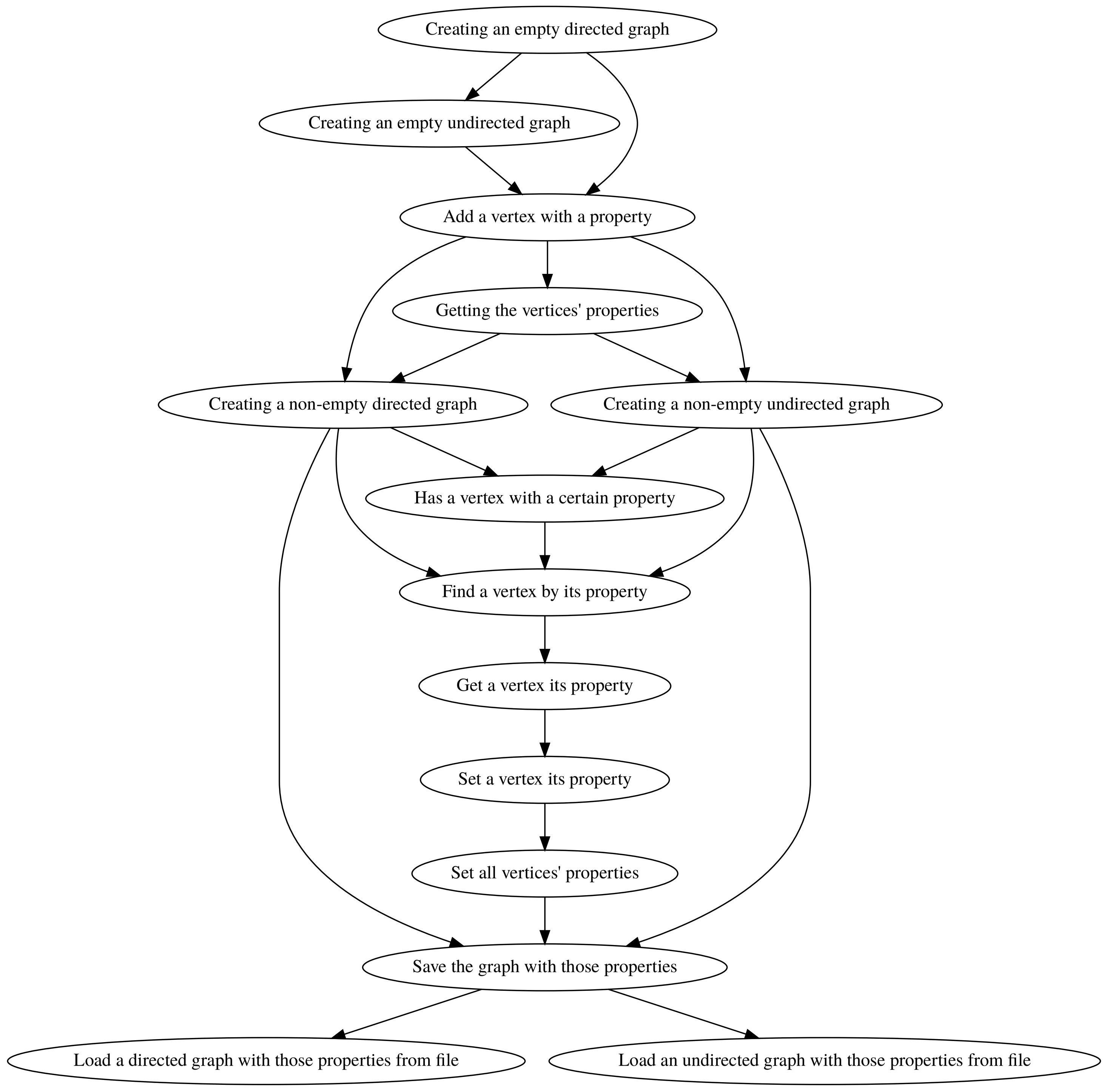

Creating an empty directed graph
Creating an empty undirected graph
Add a vertex with a property
Getting the vertices' properties
Creating a non-empty directed graph
Creating a non-empty undirected graph
Has a vertex with a certain property
Find a vertex by its property
Get a vertex its property
Set 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 2: The relations between sub-chapters

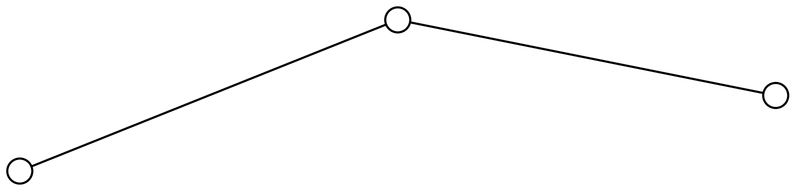


Figure 3: Example of an undirected graph

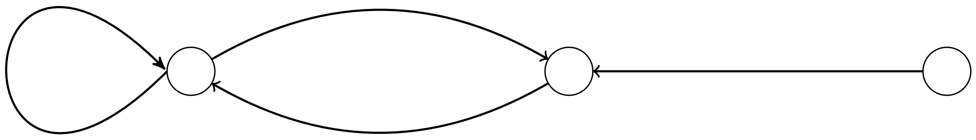


Figure 4: Example of a directed graph

From such graphs, there are some things we can do.

- [list of functions here]

2.1 Create a direct-neighbour subgraph from a vertex descriptor

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

Here is the 'create_direct_neighbour_subgraph' code:

Algorithm 1 Get the direct-neighbour subgraph from a vertex descriptor

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

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

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

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

Algorithm 2 Demo of the 'create_direct_neighbour_subgraph' function

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

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

Note that this algorithm works on both undirected and directional graphs. If the graph is directional, only the out edges will be copied. To also copy the vertices connected with inward edges, use 2.2

2.2 Create a direct-neighbour subgraph from a vertex descriptor including inward edges

Too bad, this algorithm does not work yet.

Algorithm 3 Get the direct-neighbour subgraph from a vertex descriptor

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

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

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

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

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

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

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

2.3 Creating all direct-neighbour subgraphs from a graph without properties

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

Algorithm 4 Create all direct-neighbour subgraphs from a graph without properties

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

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

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

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

Algorithm 5 Demo of the 'create_all_direct_neighbour_subgraphs' function

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

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

2.4 Count the number of connected components in an directed graph

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

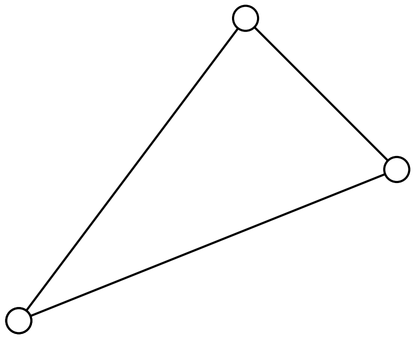
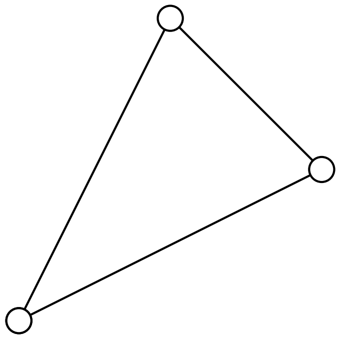



Figure 5: Example of a directed graph with two components

This algorithm counts the number of connected components:

Algorithm 6 Count the number of connected components

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

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

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

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

Algorithm 7 Demo of the 'count_directed_graph_connected_components' function

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

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

2.5 Count the number of connected components in an undirected graph

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

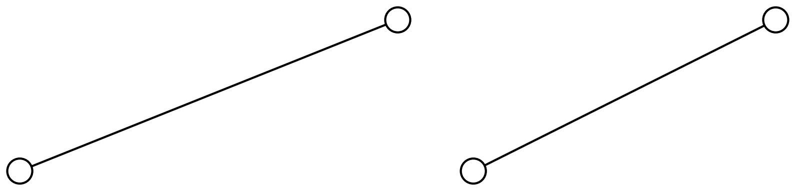


Figure 6: Example of an undirected graph with two components

This algorithm counts the number of connected components:

Algorithm 8 Count the number of connected components

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

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

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

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

Algorithm 9 Demo of the 'count_undirected_graph_connected_components' function

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

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

2.6 Count the number of levels in an undirected graph

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

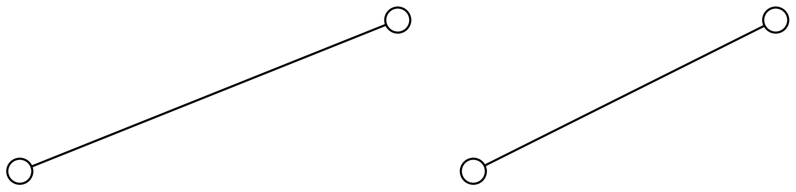


Figure 7: Example of an undirected graph with two components

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

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

Algorithm 10 Count the number of levels in an undirected graph

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

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

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

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

    const auto t = s;

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

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

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

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

Algorithm 11 Demo of the 'count_undirected_graph_levels' function

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

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

3 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].

3.1 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.

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

Here I will show the header file of 'my_bundled_vertex', as the implementation of it is not important:

Algorithm 12 Declaration of my_bundled_vertex

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

struct my_bundled_vertex
{
  explicit my_bundled_vertex(
    const std::string& name = "",
    const std::string& description = "",
    const double x = 0.0,
    const double y = 0.0
  ) noexcept;
  std::string m_name;
  std::string m_description;
  double m_x;
  double m_y;
};

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

'my_bundled_vertex' is a class that has multiple properties:

- It has four public member variables: the double 'm_x' ('m_' stands for member), the double 'm_y', the std::string m_name and the std::string m_description. These variables must be public
- It has a default constructor
- It is copyable
- It is comparable for equality (it has operator==), which is needed for searching

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

3.2 [Algorithms on graphs of bundled vertices]

[Algorithms on graphs of bundled vertices]

4 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[2].

4.1 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.

Algorithm 13 Declaration of my_bundled_edge

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

class my_bundled_edge
{
public:
  explicit my_bundled_edge(
    const std::string& name = "",
    const std::string& description = "",
    const double width = 1.0,
    const double height = 1.0
  ) noexcept;
  std::string m_name;
  std::string m_description;
  double m_width;
  double m_height;
};

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

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

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.

4.2 [some algorithms]

Some algorithms

References

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

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

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

Index