31. Boost.Graph

Boost.Graph provides tools to work with graphs. Graphs are two-dimensional point clouds with any number of lines between points. A subway map is a good example of a graph. Subway stations are points, which are connected by subway lines.

The graph theory is the field of mathematics that researches graphs. Graph theory tries to answer questions such as how to determine the shortest path between two points. Auto navigation systems have to solve that problem to guide drivers to their desired location using the shortest path. Graphs are very important in practice because many problems can be modelled with them.

Boost.Graph provides containers to define graphs. However, even more important are the algorithms Boost.Graph offers to operate on graphs, for example, to find the shortest path. This chapter introduces you to the containers and algorithms in Boost.Graph.

Vertices and Edges

Graphs consist of points and lines. To create a graph, you have to define a set of points and any lines between them. Example 31.1, “A graph of type boost::adjacency_list with four vertices” contains a first simple graph consisting of four points and no lines.

Boost.Graph provides three containers to define graphs. The most important container is `boost::adjacency_list` which is used in nearly all of the examples in this chapter. To use this class, include the header file `boost/graph/adjacency_list.hpp`. If you want to use another container, you must include another header file. There is no master header file to get access to all classes and functions from Boost.Graph.

Example 31.1. A graph of type `boost::adjacency_list` with four vertices

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

int main()
{
  boost::adjacency_list<> g;

  boost::adjacency_list<>::vertex_descriptor v1 = boost::add_vertex(g);
  boost::adjacency_list<>::vertex_descriptor v2 = boost::add_vertex(g);
  boost::adjacency_list<>::vertex_descriptor v3 = boost::add_vertex(g);
  boost::adjacency_list<>::vertex_descriptor v4 = boost::add_vertex(g);

  std::cout << v1 << ", " << v2 << ", " << v3 << ", " << v4 << '\n';
}
```

`boost::adjacency_list` is a template that is instantiated with default parameters in Example 31.1, “A graph of type `boost::adjacency_list` with four vertices”. Later, you will see what parameters you can pass. This class is defined in `boost`. All classes and functions from Boost.Graph are defined in this namespace.

To add four points to the graph, the function `boost::add_vertex()` has to be called four times.

`boost::add_vertex()` is a free-standing function and not a member function of `boost::adjacency_list`. You will find there are many free-standing functions in Boost.Graph that could have been implemented as member functions. Boost.Graph is designed to be more of a generic library than an object-oriented library.

`boost::add_vertex()` adds a point to a graph. In graph theory, a point is called vertex, which explains the function name.

`boost::add_vertex()` returns an object of type `boost::adjacency_list::vertex_descriptor`. This object represents a newly added point in the graph. You can write the objects to standard output as shown in Example 31.1, "A graph of type `boost::adjacency_list` with four vertices". The example displays `0, 1, 2, 3`.

Example 31.1, "A graph of type `boost::adjacency_list` with four vertices" identifies points through positive integers. These numbers are indexes to a vector that is used internally in `boost::adjacency_list`. It's no surprise that `boost::add_vertex()` returns 0, 1, 2, and 3 since every call adds another point to the vector.

`std::vector` is the container `boost::adjacency_list` uses by default to store points. In this case, `boost::adjacency_list::vertex_descriptor` is a type definition for `std::size_t`. Because other containers can be used to store points, `boost::adjacency_list::vertex_descriptor` isn't necessarily always `std::size_t`.

Example 31.2. Accessing vertices with `boost::vertices()`

```
#include <boost/graph/adjacency_list.hpp>
#include <utility>
#include <algorithm>
#include <iterator>
#include <iostream>

int main()
{
  boost::adjacency_list<> g;

  boost::add_vertex(g);
  boost::add_vertex(g);
  boost::add_vertex(g);
  boost::add_vertex(g);

  std::pair<boost::adjacency_list<>::vertex_iterator,
    boost::adjacency_list<>::vertex_iterator> vs = boost::vertices(g);

  std::copy(vs.first, vs.second,
    std::ostream_iterator<boost::adjacency_list<>::vertex_descriptor>{
      std::cout, "\n"});
}
```

To get all points from a graph, call `boost::vertices()`. This function returns two iterators of type `boost::adjacency_list::vertex_iterator`, which refer to the beginning and ending points. The iterators are returned in a `std::pair`. Example 31.2, "Accessing vertices with `boost::vertices()`" uses the iterators to write all points to standard output. This example displays the number 0, 1, 2, and 3, just like the previous example.

Example 31.3, "Accessing edges with `boost::edges()`" explains how points are connected with lines.

You call `boost::add_edge()` to connect two points in a graph. You have to pass the points and the graph as parameters. In graph theory, lines between points are called edges – that's why the function is called `boost::add_edge()`.

`boost::add_edge()` returns a `std::pair`. **`first`** provides access to the line. **`second`** is a `bool` variable that indicates whether the line was successfully added. If you run Example 31.3, "Accessing edges with `boost::edges()`", you'll see that `p.second` is set to `true` for each call to `boost::add_edge()`, and a new line is added to the graph with each call.

`boost::edges()` provides access to all lines in a graph. Like `boost::vertices()`, `boost::edges()` returns two iterators that refer to the beginning and ending lines. Example 31.3, "Accessing edges with `boost::edges()`" writes all lines to standard output. The example displays `(0,1)`, `(0,1)` and `(1,0)`.

Example 31.3. Accessing edges with `boost::edges()`

```
#include <boost/graph/adjacency_list.hpp>
#include <utility>
#include <algorithm>
#include <iterator>
#include <iostream>

int main()
{
  boost::adjacency_list<> g;

  boost::adjacency_list<>::vertex_descriptor v1 = boost::add_vertex(g);
  boost::adjacency_list<>::vertex_descriptor v2 = boost::add_vertex(g);
  boost::add_vertex(g);
  boost::add_vertex(g);

  std::pair<boost::adjacency_list<>::edge_descriptor, bool> p =
    boost::add_edge(v1, v2, g);
  std::cout.setf(std::ios::boolalpha);
  std::cout << p.second << '\n';
  p = boost::add_edge(v1, v2, g);
  std::cout << p.second << '\n';

  p = boost::add_edge(v2, v1, g);
  std::cout << p.second << '\n';

  std::pair<boost::adjacency_list<>::edge_iterator,
    boost::adjacency_list<>::edge_iterator> es = boost::edges(g);

  std::copy(es.first, es.second,
    std::ostream_iterator<boost::adjacency_list<>::edge_descriptor>{
      std::cout, "\n"});
}
```

The output shows that the graph has three lines. All three connect the first two points – those with the indexes 0 and 1. The output also shows where the lines start and end. Two lines start at the first point, one at the second. The direction of the lines depends on the order of the parameters passed to `boost::add_edge()`.

As you see, you can have multiple lines between the same two points. However, this feature can be deactivated.

Example 31.4, "`boost::adjacency_list` with selectors" doesn't instantiate `boost::adjacency_list` with default template parameters. Three parameters, called selectors, are passed in. By convention, the names of selectors end in S. These selectors determine what types will be used in `boost::adjacency_list` to store points and lines.

Example 31.4. `boost::adjacency_list` with selectors

```
#include <boost/graph/adjacency_list.hpp>
#include <utility>
#include <algorithm>
#include <iterator>
#include <iostream>

int main()
{
  typedef boost::adjacency_list<boost::setS, boost::vecS,
    boost::undirectedS> graph;
  graph g;

  boost::adjacency_list<>::vertex_descriptor v1 = boost::add_vertex(g);
  boost::adjacency_list<>::vertex_descriptor v2 = boost::add_vertex(g);
  boost::add_vertex(g);
  boost::add_vertex(g);

  std::pair<graph::edge_descriptor, bool> p =
    boost::add_edge(v1, v2, g);
  std::cout.setf(std::ios::boolalpha);
  std::cout << p.second << '\n';
  p = boost::add_edge(v1, v2, g);
  std::cout << p.second << '\n';

  p = boost::add_edge(v2, v1, g);
  std::cout << p.second << '\n';

  std::pair<graph::edge_iterator,
    graph::edge_iterator> es = boost::edges(g);

  std::copy(es.first, es.second,
    std::ostream_iterator<graph::edge_descriptor>{std::cout, "\n"});
}
```

By default, `boost::adjacency_list` uses `std::vector` for points and lines. By passing `boost::setS` as the first template parameter in Example 31.4, "`boost::adjacency_list` with selectors", `std::set` is selected as the container for lines. Because `std::set` doesn't support duplicates, it is not possible to add the same line using `boost::add_edge()` multiple times. Thus, the example only displays `(0,1)` once.

The second template parameter tells `boost::adjacency_list` which class should be used for points. In Example 31.4, "`boost::adjacency_list` with selectors", `boost::vecS` is passed. This is the default value for the second template parameter. It is only set so that you can pass a third template parameter.

The third template parameter determines whether lines are directed or undirected. The default is `boost::directedS`, which means all lines are directed and can be drawn as arrows. Lines can only be crossed in one direction.

`boost::undirectedS` is used in Example 31.4, "`boost::adjacency_list` with selectors". This selector makes all lines undirected, which means it is possible to cross a line in any direction. It doesn't matter which point is the start and which is the end. This is another reason why the graph in Example 31.4, "`boost::adjacency_list` with selectors" contains only one line. The third call to the function `boost::add_edge()` swaps the start and end points, but because lines in this example are undirected, this line is the same as the previous lines and, therefore, isn't added.

Boost.Graph offers more selectors, including `boost::listS`, `boost::mapS`, and `boost::hash_setS`. `boost::bidirectionalS` can be used to make lines bidirectional. This selector is similar to `boost::undirectedS`, but in this case, start and end points matter. If you use `boost::bidirectionalS` in Example 31.4, "`boost::adjacency_list` with selectors", the third call to `boost::add_edge()` will add a line to the graph.

Example 31.5, "Creating indexes automatically with `boost::add_edge()`" shows a simpler method for adding points and lines to a graph.

Example 31.5. Creating indexes automatically with `boost::add_edge()`

```
#include <boost/graph/adjacency_list.hpp>
#include <tuple>
#include <algorithm>
#include <iterator>
#include <iostream>

int main()
{
  typedef boost::adjacency_list<boost::setS, boost::vecS,
    boost::undirectedS> graph;
  graph g;

  enum { topLeft, topRight, bottomRight, bottomLeft };

  boost::add_edge(topLeft, topRight, g);
  boost::add_edge(topRight, bottomRight, g);
  boost::add_edge(bottomRight, bottomLeft, g);
  boost::add_edge(bottomLeft, topLeft, g);

  graph::edge_iterator it, end;
  std::tie(it, end) = boost::edges(g);
  std::copy(it, end,
    std::ostream_iterator<graph::edge_descriptor>{std::cout, "\n"});
}
```

Example 31.5, "Creating indexes automatically with `boost::add_edge()`" defines a graph consisting of four points. You can visualize the graph as a map with four fields, each represented by a point. The points are given the names `topLeft`, `topRight`, `bottomRight`, and `bottomLeft`. Because the names are assigned in an enumeration, each will have a numeric value that is used as an index.

It is possible to define a graph without calling `boost::add_vertex()`. Boost.Graph adds missing points to a graph automatically if the points passed to `boost::add_edge()` don't exist. The multiple calls to `boost::add_edge()` in Example 31.5, "Creating indexes automatically with `boost::add_edge()`" define not only lines but also add the four points required for the lines to the graph.

Please note how `std::tie()` is used to store the iterators returned in a `std::pair` from `boost::edges()` in **`it`** and **`end`**. `std::tie()` has been part of the standard library since C++11.

The graph in Example 31.5, "Creating indexes automatically with `boost::add_edge()`" is a map with four fields. To get from the top left to the bottom right, one can either cross the field in the top right or the one in the bottom left. There is no line between opposite fields. Thus it’s not possible to go directly from the top left to the bottom right. All examples in this chapter use this graph.

Example 31.6. `boost::adjacent_vertices()` and `boost::out_edges()`

```
#include <boost/graph/adjacency_list.hpp>
#include <tuple>
#include <algorithm>
#include <iterator>
#include <iostream>

int main()
{
  typedef boost::adjacency_list<boost::setS, boost::vecS,
    boost::undirectedS> graph;
  graph g;

  enum { topLeft, topRight, bottomRight, bottomLeft };

  boost::add_edge(topLeft, topRight, g);
  boost::add_edge(topRight, bottomRight, g);
  boost::add_edge(bottomRight, bottomLeft, g);
  boost::add_edge(bottomLeft, topLeft, g);

  graph::adjacency_iterator vit, vend;
  std::tie(vit, vend) = boost::adjacent_vertices(topLeft, g);
  std::copy(vit, vend,
    std::ostream_iterator<graph::vertex_descriptor>{std::cout, "\n"});

  graph::out_edge_iterator eit, eend;
  std::tie(eit, eend) = boost::out_edges(topLeft, g);
  std::for_each(eit, eend,
    [&g](graph::edge_descriptor it)
      { std::cout << boost::target(it, g) << '\n'; });
}
```

Example 31.6, “`boost::adjacent_vertices()` and `boost::out_edges()`” introduces functions to gain additional information on points. `boost::adjacent_vertices()` returns a pair of iterators that refer to points a point connects to. You call `boost::out_edges()` if you want to access all outgoing lines from a point. `boost::in_edges()` accesses all ingoing lines. With undirected lines, it doesn’t matter which of the two functions is called.

`boost::target()` returns the end point of a line. The start point is returned with `boost::source()`.

Example 31.6, “`boost::adjacent_vertices()` and `boost::out_edges()`” writes 1 and 3, the indexes of the top right and bottom left fields, to standard output twice. `boost::adjacent_vertices()`, is called with **`topLeft`** and returns and displays the indexes of the top right and bottom left fields. **`topLeft`** is also passed to `boost::out_edges()` to retrieve the outgoing lines. Because `boost::target()`

is called on every outgoing line with `std::for_each()`, the indexes of the top right and bottom left fields are displayed twice.

Example 31.7, "Initializing `boost::adjacency_list` with lines" illustrates how to define a graph with `boost::adjacency_list` without having to call `boost::add_edge()` for every line.

Example 31.7. Initializing `boost::adjacency_list` with lines

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

int main()
{
  enum { topLeft, topRight, bottomRight, bottomLeft };

  std::array<std::pair<int, int>, 4> edges{{
    std::make_pair(topLeft, topRight),
    std::make_pair(topRight, bottomRight),
    std::make_pair(bottomRight, bottomLeft),
    std::make_pair(bottomLeft, topLeft)
  }};

  typedef boost::adjacency_list<boost::setS, boost::vecS,
    boost::undirectedS> graph;
  graph g{edges.begin(), edges.end(), 4};

  std::cout << boost::num_vertices(g) << '\n';
  std::cout << boost::num_edges(g) << '\n';

  g.clear();
}
```

You can pass iterators to the constructor of `boost::adjacency_list` that refer to objects of type `std::pair<int, int>`, which define lines. If you pass iterators, you also have to supply a third parameter that determines the total number of points in the graph. The graph will contain at least the points required for the lines. The third parameter let's you add points to the graph that aren't connected to other points.

Example 31.7, "Initializing `boost::adjacency_list` with lines" uses the functions `boost::num_vertices()` and `boost::num_edges()`, which return the number of points and lines, respectively. The example displays `4` twice.

Example 31.7, "Initializing `boost::adjacency_list` with lines" calls `boost::adjacency_list::clear()`. This member function removes all points and lines. It is a member function of `boost::adjacency_list` and not a free-standing function.

Algorithms

Algorithms from Boost.Graph resemble those from the standard library – they are generic and very flexible. However, it’s not always immediately clear how they should be used.

Example 31.8. Visiting points from inside to outside with `breadth_first_search()`

```
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/breadth_first_search.hpp>
#include <boost/graph/named_function_params.hpp>
#include <boost/graph/visitors.hpp>
#include <boost/array.hpp>
#include <array>
#include <utility>
#include <iterator>
#include <algorithm>
#include <iostream>

int main()
{
  enum { topLeft, topRight, bottomRight, bottomLeft };

  std::array<std::pair<int, int>, 4> edges{{
    std::make_pair(topLeft, topRight),
    std::make_pair(topRight, bottomRight),
    std::make_pair(bottomRight, bottomLeft),
    std::make_pair(bottomLeft, topLeft)
  }};

  typedef boost::adjacency_list<boost::setS, boost::vecS,
    boost::undirectedS> graph;
  graph g{edges.begin(), edges.end(), 4};

  boost::array<int, 4> distances{{0}};

  boost::breadth_first_search(g, topLeft,
    boost::visitor(
      boost::make_bfs_visitor(
        boost::record_distances(distances.begin(),
          boost::on_tree_edge{}))));

  std::copy(distances.begin(), distances.end(),
    std::ostream_iterator<int>{std::cout, "\n"});
}
```

Example 31.8, “Visiting points from inside to outside with `breadth_first_search()`” uses the algorithm `boost::breadth_first_search()` to visit points from inside to outside. The algorithm starts at the point passed as the second parameter. It first visits all points that can be reached directly from that point, working like a wave.

`boost::breadth_first_search()` doesn't return a specific result. The algorithm just visits points. Whether data is collected and stored depends on the visitors passed to `boost::breadth_first_search()`.

Visitors are objects whose member functions are called when a point is visited. By passing visitors to an algorithm like `boost::breadth_first_search()`, you decide what should happen when a point is visited. Visitors are like function objects that can be passed to algorithms of the standard library.

Example 31.8, "Visiting points from inside to outside with `breadth_first_search()`" uses a visitor that records distances. A distance is the number of lines that have to be crossed to get from one point to another, starting at the point passed to `boost::breadth_first_search()` as the second parameter. Boost.Graph provides the helper function `boost::record_distances()` to create the visitor. A property map and a tag also have to be passed.

Property maps store properties for points or lines. Boost.Graph describes the concept of property maps. Since a pointer or iterator is taken as the beginning of a property map, it isn't important to understand property maps in detail. In Example 31.8, "Visiting points from inside to outside with `breadth_first_search()`" the beginning of the array **`distances`** is passed with `distances.begin()` to `boost::record_distances()`. This is sufficient for the array **`distances`** to be used as a property map. However, it is important that the size of the array isn't smaller than the number of points in the graph. After all, the distance to each and every point in the graph needs to be stored.

Please note that **`distances`** is based on `boost::array` and not on `std::array`. Using `std::array` would lead to a compiler error.

Depending on the algorithm, there are different events. The second parameter passed to `boost::record_distances()` specifies which events the visitor should be notified about. Boost.Graph defines tags that are empty classes to give events names. The tag `boost::on_tree_edge` in Example 31.8, "Visiting points from inside to outside with `breadth_first_search()`" specifies that a distance should be recorded when a new point has been found.

Events depend on the algorithm. You have to check the documentation on algorithms to find out which events are supported and which tags you can use.

A visitor created by `boost::record_distances()` is algorithm independent, so you can use `boost::record_distances()` with other algorithms. An adapter is used to bind an algorithm and a visitor. Example 31.8, "Visiting points from inside to outside with `breadth_first_search()`" calls `boost::make_bfs_visitor()` to create this adapter. This helper function returns a visitor as expected by the algorithm `boost::breadth_first_search()`. This visitor defines member functions that fit the events the algorithm supports. For example, the visitor returned by `boost::make_bfs_visitor()` defines the member function `tree_edge()`. If a visitor that is defined with the tag `boost::on_tree_edge` is passed to `boost::make_bfs_visitor()` (as in Example 31.8, "Visiting points from inside to outside with `breadth_first_search()`"), the visitor is notified when `tree_edge()` is called. This lets you use visitors with different algorithms without those visitors having to define all of the member functions expected by all algorithms.

The adapter returned by `boost::make_bfs_visitor()` can't be passed directly to the algorithm `boost::breadth_first_search()`. It has to be wrapped with `boost::visitor()` and then passed as a third parameter.

There are two variants of algorithms like `boost::breadth_first_search()`. One variant expects that every parameter the algorithm supports will be passed. Another variant supports something similar to named parameters. It's typically easier to use this second variant because only the parameters you're interested in have to be passed. Many parameters don't have to be passed because algorithms use default values.

Example 31.8, "Visiting points from inside to outside with `breadth_first_search()`" uses the variant of `boost::breadth_first_search()` that expects named parameters. The first two parameters are the graph and the start point, which are required. However, the third parameter can be nearly everything. In Example 31.8, "Visiting points from inside to outside with `breadth_first_search()`" a visitor needs to be passed. For that to work, the adapter returned by `boost::make_bfs_visitor()` is named using `boost::visitor()`. Now, it's clear that the third parameter is a visitor. You'll see in the following examples how other parameters are passed by name to `boost::breadth_first_search()`.

Example 31.8, "Visiting points from inside to outside with `breadth_first_search()`" displays the numbers 0, 1, 2, and 1. These are the distances to all points from the top left. The top right field – the one with the index 1 – is only one step away. The bottom right field – the one with the index 2 – is two steps away. The bottom left field – the one with the index 3 – is again only one step away. The number 0 which is printed first refers to the top left field. Since it’s the start point that was passed to `boost::breadth_first_search()`, zero steps are required to reach it.

`boost::breadth_first_search()` doesn’t set the elements in the array, it just increases the stored values. Therefore, you must initialize all elements in the array **`distances`** before you start.

Example 31.9, “Finding paths with `breadth_first_search()`” illustrates how to find the shortest path.

Example 31.9, “Finding paths with `breadth_first_search()`” displays 0, 1, and 2. This is the shortest path from top left to bottom right. It leads over the top right field although the path over the bottom left field would be equally short.

`boost::breadth_first_search()` is used again – this time to find the shortest path. As you already know, this algorithm just visits points. To get a description of the shortest path, an appropriate visitor must be used. Example 31.9, “Finding paths with `breadth_first_search()`” calls `boost::record_predecessors()` to get one.

`boost::record_predecessors()` returns a visitor to store the predecessor of every point. Whenever `boost::breadth_first_search()` visits a new point, the previous point is stored in the property map passed to `boost::record_predecessors()`. As `boost::breadth_first_search()` visits points from the inside to the outside, the shortest path is found – starting at the point passed as a second parameter to `boost::breadth_first_search()`. Example 31.9, “Finding paths with `breadth_first_search()`” finds the shortest paths from all points in the graph to the bottom right.

Example 31.9. Finding paths with `breadth_first_search()`

```
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/breadth_first_search.hpp>
#include <boost/graph/named_function_params.hpp>
#include <boost/graph/visitors.hpp>
#include <boost/array.hpp>
#include <array>
#include <utility>
#include <algorithm>
#include <iostream>

int main()
{
  enum { topLeft, topRight, bottomRight, bottomLeft };

  std::array<std::pair<int, int>, 4> edges{{
    std::make_pair(topLeft, topRight),
    std::make_pair(topRight, bottomRight),
    std::make_pair(bottomRight, bottomLeft),
    std::make_pair(bottomLeft, topLeft)
  }};

  typedef boost::adjacency_list<boost::setS, boost::vecS,
    boost::undirectedS> graph;
  graph g{edges.begin(), edges.end(), 4};

  boost::array<int, 4> predecessors;
  predecessors[bottomRight] = bottomRight;

  boost::breadth_first_search(g, bottomRight,
    boost::visitor(
      boost::make_bfs_visitor(
        boost::record_predecessors(predecessors.begin(),
          boost::on_tree_edge{}))));

  int p = topLeft;
  while (p != bottomRight)
  {
    std::cout << p << '\n';
    p = predecessors[p];
  }
  std::cout << p << '\n';
}
```

After `boost::breadth_first_search()` returns, the property map **`predecessors`** contains the predecessor of every point. To find the first field when travelling from the top left to the bottom right, the element with the index 0 – the index of the top left field – is accessed in **`predecessors`**. The value found in **`predecessors`** is 1, which means the next field is at the top right. Accessing **`predecessors`** with the index 1 returns the next field. In Example 31.9, "Finding paths with `breadth_first_search()`" that's the bottom right field – the one with the index 2. That way it's possible to find the points iteratively in huge graphs to get from a start to an end point.

Example 31.10. Finding distances and paths with `breadth_first_search()`

```
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/breadth_first_search.hpp>
#include <boost/graph/named_function_params.hpp>
#include <boost/graph/visitors.hpp>
#include <boost/array.hpp>
#include <array>
#include <utility>
#include <algorithm>
#include <iostream>

int main()
{
  enum { topLeft, topRight, bottomRight, bottomLeft };

  std::array<std::pair<int, int>, 4> edges{{
    std::make_pair(topLeft, topRight),
    std::make_pair(topRight, bottomRight),
    std::make_pair(bottomRight, bottomLeft),
    std::make_pair(bottomLeft, topLeft)
  }};

  typedef boost::adjacency_list<boost::setS, boost::vecS,
    boost::undirectedS> graph;
  graph g{edges.begin(), edges.end(), 4};

  boost::array<int, 4> distances{{0}};
  boost::array<int, 4> predecessors;
  predecessors[bottomRight] = bottomRight;

  boost::breadth_first_search(g, bottomRight,
    boost::visitor(
      boost::make_bfs_visitor(
        std::make_pair(
          boost::record_distances(distances.begin(),
            boost::on_tree_edge()),
          boost::record_predecessors(predecessors.begin(),
            boost::on_tree_edge{})))));

  std::for_each(distances.begin(), distances.end(),
    [](int d){ std::cout << d << '\n'; });

  int p = topLeft;
  while (p != bottomRight)
  {
    std::cout << p << '\n';
    p = predecessors[p];
  }
  std::cout << p << '\n';
}
```

Example 31.10, "Finding distances and paths with `breadth_first_search()`" shows how `boost::breadth_first_search()` is used with two visitors. To use two visitors, you need to put them in a pair with `std::make_pair()`. If more than two visitors are needed, the pairs have to be nested. Example 31.10, "Finding distances and paths with `breadth_first_search()`" does the same thing as Example 31.8, "Visiting points from inside to outside with `breadth_first_search()`" and Example 31.9, "Finding paths with `breadth_first_search()`" together.

`boost::breadth_first_search()` can only be used if every line has the same weight. This means the time taken to cross any line between points is always the same. If lines are weighted, meaning that each line may require a different amount of time to traverse, then you need to use a different algorithm to find the shortest path.

Example 31.11. Finding paths with `dijkstra_shortest_paths()`

```
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/dijkstra_shortest_paths.hpp>
#include <boost/graph/named_function_params.hpp>
#include <boost/array.hpp>
#include <array>
#include <utility>
```

```
#include <utility>
#include <iostream>

int main()
{
  enum { topLeft, topRight, bottomRight, bottomLeft };

  std::array<std::pair<int, int>, 4> edges{{
    std::make_pair(topLeft, topRight),
    std::make_pair(topRight, bottomRight),
    std::make_pair(bottomRight, bottomLeft),
    std::make_pair(bottomLeft, topLeft)
  }};

  typedef boost::adjacency_list<boost::listS, boost::vecS,
    boost::undirectedS, boost::no_property,
    boost::property<boost::edge_weight_t, int>> graph;

  std::array<int, 4> weights{{2, 1, 1, 1}};

  graph g{edges.begin(), edges.end(), weights.begin(), 4};

  boost::array<int, 4> directions;
  boost::dijkstra_shortest_paths(g, bottomRight,
    boost::predecessor_map(directions.begin()));

  int p = topLeft;
  while (p != bottomRight)
  {
    std::cout << p << '\n';
    p = directions[p];
  }
  std::cout << p << '\n';
}
```

Example 31.11, "Finding paths with `dijkstra_shortest_paths()`" uses `boost::dijkstra_shortest_paths()` to find the shortest paths to the bottom right. This algorithm is used if lines are weighted. Example 31.11, "Finding paths with `dijkstra_shortest_paths()`" assumes that it takes twice as long to cross the line from the top left to the top right as it takes to cross any other line.

Before `boost::dijkstra_shortest_paths()` can be used, weights have to be assigned to lines. This is done with the array **`weights`**. The elements in the array correspond to the lines in the graph. Because the line from the top left to the top right is first, the first element in **`weights`** is set to a value twice as big as all others.

To assign weights to lines, the iterator to the beginning of the array **`weights`** is passed as the third parameter to the constructor of the graph. This third parameter can be used to initialize properties of lines. This only works if properties have been defined for lines.

Example 31.11, "Finding paths with `dijkstra_shortest_paths()`" passes additional template parameters to `boost::adjacency_list`. The fourth and fifth template parameter specify if points and lines have properties and what those properties are. You can assign properties to both lines and points.

By default, `boost::adjacency_list` uses `boost::no_property`, which means that neither points nor lines have properties. In Example 31.11, "Finding paths with `dijkstra_shortest_paths()`", `boost::no_property` is passed as a fourth parameter to specify no properties for points. The fifth parameter uses `boost::property` to define a bundled property.

Bundled properties are properties that are stored internally in a graph. Because it's possible to define multiple bundled properties, `boost::property` expects a tag to define each property. Boost.Graph provides some tags, such as `boost::edge_weight_t`, to define frequently used properties that are automatically recognized and used by algorithms. The second template parameter passed to `boost::property` is the type of the property. In Example 31.11, "Finding paths with `dijkstra_shortest_paths()`" weights are `int` values.

Example 31.11, "Finding paths with `dijkstra_shortest_paths()`" works because `boost::dijkstra_shortest_paths()` automatically uses the bundled property of type `boost::edge_weight_t`.

Note that no visitor is passed to `boost::dijkstra_shortest_paths()`. This algorithm doesn't just visit points. It looks for shortest paths – that's why it's called `boost::dijkstra_shortest_paths()`. You don't need to think about events or visitors. You only need to pass a container to store the predecessor of every point. If you use the variant of `boost::dijkstra_shortest_paths()` that expects named parameters, as in Example 31.11, "Finding paths with `dijkstra_shortest_paths()`", pass the container with `boost::predecessor_map()`. This is a helper function which expects a pointer or an iterator to the beginning of an array.

Example 31.11, "Finding paths with `dijkstra_shortest_paths()`" displays 0, 3, and 2: The shortest path from top left to bottom right leads over the bottom left field. The path over the top right field has a greater weight than the other possibilities.

Example 31.12. User-defined properties with `dijkstra_shortest_paths()`

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

```
#include <boost/graph/named_function_params.hpp>
#include <boost/array.hpp>
#include <array>
#include <utility>
#include <iostream>

int main()
{
  enum { topLeft, topRight, bottomRight, bottomLeft };

  std::array<std::pair<int, int>, 4> edges{{
    std::make_pair(topLeft, topRight),
    std::make_pair(topRight, bottomRight),
    std::make_pair(bottomRight, bottomLeft),
    std::make_pair(bottomLeft, topLeft)
  }};

  struct edge_properties
  {
    int weight;
  };

  typedef boost::adjacency_list<boost::listS, boost::vecS,
    boost::undirectedS, boost::no_property,
    edge_properties> graph;

  graph g{edges.begin(), edges.end(), 4};

  graph::edge_iterator it, end;
  boost::tie(it, end) = boost::edges(g);
  g[*it].weight = 2;
  g[*++it].weight = 1;
  g[*++it].weight = 1;
  g[*++it].weight = 1;

  boost::array<int, 4> directions;
  boost::dijkstra_shortest_paths(g, bottomRight,
    boost::predecessor_map(directions.begin()).
    weight_map(boost::get(&edge_properties::weight, g)));

  int p = topLeft;
  while (p != bottomRight)
  {
    std::cout << p << '\n';
    p = directions[p];
  }
  std::cout << p << '\n';
}
```

Example 31.12, "User-defined properties with `dijkstra_shortest_paths()`" works like the previous one and displays the same numbers, but it uses a user-defined class, `edge_properties`, rather than a predefined property.

`edge_properties` defines the member variable **`weight`** to store the weight of a line. It is possible to add more member variables if other properties are required.

You can access user-defined properties if you use the descriptor of lines as an index for the graph. Thus, the graph behaves like an array. You get the descriptors from the line iterators that are returned from `boost::edges()`. That way a weight can be assigned to every line.

To make `boost::dijkstra_shortest_paths()` understand that weights are stored in **`weight`** in `edge_properties`, another named parameter has to be passed. This is done with `weight_map()`. Note that `weight_map()` is a member function of the object returned from `boost::predecessor_map()`. There is also a free-standing function called `boost::weight_map()`. If you need to pass multiple named parameters, you have to call a member function on the first named parameter (the one that was returned by the free-standing function). That way all parameters are packed into one object that is then passed to the algorithm.

To tell `boost::dijkstra_shortest_paths()` that **`weight`** in `edge_properties` contains the weights, a pointer to that property is passed. It isn’t passed to `weight_map()` directly. Instead it is passed in an object created with `boost::get()`. Now the call is complete, and `boost::dijkstra_shortest_paths()` knows which property to access to get the weights.

Example 31.13. Initializing user-defined properties at graph definition

```
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/dijkstra_shortest_paths.hpp>
#include <boost/graph/named_function_params.hpp>
#include <boost/array.hpp>
#include <array>
#include <utility>
#include <iostream>

int main()
{
  enum { topLeft, topRight, bottomRight, bottomLeft };

  std::array<std::pair<int, int>, 4> edges{{
    std::make_pair(topLeft, topRight),
    std::make_pair(topRight, bottomRight),
    std::make_pair(bottomRight, bottomLeft),
    std::make_pair(bottomLeft, topLeft)
  }};

  struct edge_properties
  {
    int weight;
  };

  typedef boost::adjacency_list<boost::listS, boost::vecS,
    boost::undirectedS, boost::no_property,
    edge_properties> graph;

  boost::array<edge_properties, 4> props{{2, 1, 1, 1}};
  graph g{edges.begin(), edges.end(), props.begin(), 4};

  boost::array<int, 4> directions;
  boost::dijkstra_shortest_paths(g, bottomRight,
    boost::predecessor_map(directions.begin()).
    weight_map(boost::get(&edge_properties::weight, g)));

  int p = topLeft;
  while (p != bottomRight)
  {
    std::cout << p << '\n';
    p = directions[p];
  }
  std::cout << p << '\n';
}
```

It’s possible to initialize user-defined properties when a graph is defined. You only have to pass an iterator as the third parameter to the constructor of `boost::adjacency_list`, which refers to objects of the type of the user-defined property. Thus, you don’t need to access properties of lines through descriptors. Example 31.13, “Initializing user-defined properties at graph definition” works like the previous one and displays the same result.

Example 31.14. Random paths with `random_spanning_tree()`

```
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/random_spanning_tree.hpp>
#include <boost/graph/named_function_params.hpp>
#include <boost/array.hpp>
#include <array>
#include <utility>
#include <random>
#include <iostream>
#include <ctime>
#include <cstdint>

int main()
{
  enum { topLeft, topRight, bottomRight, bottomLeft };

  std::array<std::pair<int, int>, 4> edges{{
    std::make_pair(topLeft, topRight),
    std::make_pair(topRight, bottomRight),
    std::make_pair(bottomRight, bottomLeft),
    std::make_pair(bottomLeft, topLeft)
  }};

  struct edge_properties
  {
    int weight;
  };

  typedef boost::adjacency_list<boost::listS, boost::vecS,
    boost::undirectedS> graph;

  graph g{edges.begin(), edges.end(), 4};
  boost::array<int, 4> predecessors;

  std::mt19937 gen{static_cast<uint32_t>(std::time(0))};
  boost::random_spanning_tree(g, gen,
    boost::predecessor_map(predecessors.begin()).
    root_vertex(bottomLeft));

  int p = topRight;
  while (p != -1)
  {
    std::cout << p << '\n';
    p = predecessors[p];
  }
}
```

The algorithm introduced in Example 31.14, "Random paths with `random_spanning_tree()`" finds random paths. `boost::random_spanning_tree()` is similar to `boost::dijkstra_shortest_paths()`. It returns the predecessors of points in a container that is passed with `boost::predecessor_map`. In contrast to `boost::dijkstra_shortest_paths()`, the starting point isn't passed directly as a parameter to `boost::random_spanning_tree()`. It must be passed as a named parameter. That's why `root_vertex()` is called on the object of type `boost::predecessor_map`. Example 31.14, "Random paths with `random_spanning_tree()`" finds random paths to the bottom left field.

Because `boost::random_spanning_tree()` is looking for a random path, a random number generator has to be passed as the second parameter. Example 31.14, "Random paths with `random_spanning_tree()`" uses `std::mt19937`, which has been part of the standard library since C++11. You could also use a random number generator from Boost.Random.

Example 31.14, "Random paths with `random_spanning_tree()`" displays either 1, 0, and 3 or 1, 2, and 3. 1 is the top right field, 3 the bottom left field. There are only two possible paths from the top right field to the bottom left field: through the top left field or through the bottom right field. `boost::random_spanning_tree()` must return one of these two paths.

Containers

All examples in this chapter so far have used `boost::adjacency_list` to define graphs. This section introduces the two other graph containers provided by Boost.Graph: `boost::adjacency_matrix` and `boost::compressed_sparse_row_graph`.

Note

There is a missing include in `boost/graph/adjacency_matrix.hpp` in Boost 1.56.0. To compile Example 31.15, "Graphs with `boost::adjacency_matrix`" with Boost 1.56.0, include `boost/functional/hash.hpp` before `boost/graph/adjacency_matrix.hpp`.

Example 31.15. Graphs with `boost::adjacency_matrix`

```
#include <boost/graph/adjacency_matrix.hpp>
#include <array>
#include <utility>

int main()
{
  enum { topLeft, topRight, bottomRight, bottomLeft };

  std::array<std::pair<int, int>, 4> edges{{
    std::make_pair(topLeft, topRight),
    std::make_pair(topRight, bottomRight),
    std::make_pair(bottomRight, bottomLeft),
    std::make_pair(bottomLeft, topLeft)
  }};

  typedef boost::adjacency_matrix<boost::undirectedS> graph;
  graph g{edges.begin(), edges.end(), 4};
}
```

`boost::adjacency_matrix` is used like `boost::adjacency_list` (see Example 31.15, "Graphs with `boost::adjacency_matrix`"). However, the two template parameters that pass selectors don't exist with `boost::adjacency_matrix`. With `boost::adjacency_matrix`, no selectors, such as `boost::vecS` and `boost::setS`, are used. `boost::adjacency_matrix` stores the graph in a matrix, and the internal structure is hardcoded. You can think of the matrix as a two-dimensional table: the table is a square with as many rows and columns as the graph has points. A line is created by marking the cell where the row and column that correspond with the two end points of the line intersect.

The internal structure of `boost::adjacency_matrix` makes it possible to add and remove lines quickly. However, memory consumption is higher. The rule of thumb is to use `boost::adjacency_list` when there are relatively few lines compared to points. The more lines there are, the more it makes sense to use `boost::adjacency_matrix`.

Example 31.16. Graphs with `boost::compressed_sparse_row_graph`

```
#include <boost/graph/compressed_sparse_row_graph.hpp>
#include <array>
#include <utility>

int main()
{
  enum { topLeft, topRight, bottomRight, bottomLeft };

  std::array<std::pair<int, int>, 4> edges{{
    std::make_pair(topLeft, topRight),
    std::make_pair(topRight, bottomRight),
    std::make_pair(bottomRight, bottomLeft),
    std::make_pair(bottomLeft, topLeft)
  }};

  typedef boost::compressed_sparse_row_graph<boost::bidirectionalS> graph;
  graph g{boost::edges_are_unsorted_multi_pass, edges.begin(),
    edges.end(), 4};
}
```

`boost::compressed_sparse_row_graph` is used in the same way as `boost::adjacency_list` and `boost::adjacency_matrix` (see Example 31.16, “Graphs with `boost::compressed_sparse_row_graph`”). The most important difference is that graphs can’t be changed with `boost::compressed_sparse_row_graph`. Once the graph has been created, points and lines can’t be added or removed. Thus, `boost::compressed_sparse_row_graph` makes only sense when using an immutable graph.

`boost::compressed_sparse_row_graph` only supports directed lines. You can’t instantiate `boost::compressed_sparse_row_graph` with the template parameter `boost::undirectedS`.

The main advantage of `boost::compressed_sparse_row_graph` is low memory consumption. `boost::compressed_sparse_row_graph` is especially useful if you have a huge graph and you need to keep memory consumption low.