

# AlgoLab BGL Introduction

Andreas Bärtschi

October 12, 2016<sup>1</sup>

---

<sup>1</sup>Updated on October 14, 2016



A **generic** C++ library of graph data structures and algorithms.

**BGL docs** – your new best friend:

[http://www.boost.org/doc/libs/1\\_58\\_0/libs/graph/doc](http://www.boost.org/doc/libs/1_58_0/libs/graph/doc)

Moodle: There's a brief **copy & paste manual**.

Algolab VM & General: There's a [technical instructions page](#) for all things Algolab.

# BGL: A generic library

Genericity type	STL	BGL
Algorithm / Data-Structure Interoperability	Decoupling of algorithms and data-structures Key ingredients: iterators	Decoupling of graph algorithms and graph representations Vertex iterators, edge iterators, adjacency iterators
Parameterization	Element type parameterization	Vertex and edge property multi-parametrization <i>Associate multiple properties</i> <i>Accessible via property maps</i>
Extensions	through function objects	through a <i>visitor object</i> , event points and methods depend on particular algorithm

# BGL: Graph Representations / Data Structures

Structure	Representation	Advantages	Do
Graph classes	Adjacency list	Swiss army knife: Directed/undirected graphs, allow/disallow parallel-edges, efficient insertion, fast adjacency structure exploitation	<b>use this!</b>
	Adjacency matrix	Dense graphs	<i>use at your</i>
Adaptors	Edge list	Simplicity	<i>own risk!</i>
	External adaptation	Convert existing graph structures (LEDA etc.) to BGL	Not covered in Algotlab.

Example **without** any vertex or edge properties:

```
1 // Easy syntax. Parameters:
2 // OutEdgeList type, VertexList type, Directivity
3 typedef adjacency_list<vecS, vecS, directedS>      Graph;
4
5 // which is the same as:
6 typedef adjacency_list<vecS, vecS, directedS,
7     no_property,      // the graph has no interior vertex properties
8     no_property       // the graph has no interior vertex properties
9     >               Graph;
```

Defines a *directed* Graph where the vertices are stored in a vector (VertexList vecS) and the outgoing edges in each vertex are stored in a vector (OutEdgeList vecS). (Also see *Useful stuff: Options for adjacency\_list*, page 40.)

## BGL: adjacency\_list

Example **with** vertex property and multiple edge properties:

```
1 // Note the nested syntax for defining more than one edge property
2 typedef adjacency_list<vecS, vecS, directedS,
3     property<vertex_name_t, string>,           // interior vertex property
4     property<edge_capacity_t, int>,             // interior edge properties
5     property<edge_residual_capacity_t, int>,    // nested syntax
6     property<edge_reverse_t, Traits::edge_descriptor>>>> Graph;
7
8 typedef property_map<Graph, vertex_name_t>::type NameMap;
9 typedef property_map<Graph, edge_capacity_t>::type CapacityMap;
10 typedef property_map<Graph, edge_residual_capacity_t>::type ResidualMap;
11 typedef property_map<Graph, edge_reverse_t>::type ReverseMap;
```

Interior properties are stored with the graph. Property Maps allow us to access the interior properties of the graph. Think of these as a map (object with operator []). Also see *Useful stuff: Interior property maps*, pages 44–45.

# BGL: Graph Algorithms

Area	Topic	Details
Basics	Distances	Dijkstra shortest paths Prim minimum spanning tree Kruskal minimum spanning tree
	Components	Connected, biconnected & strongly connected components
	General Matchings	General unweighted matching
Flows	Maximum Flow	Graph setup (residual graph) Edmonds-Karp and Push-Relabel
	Disjoint paths	Vertex- / Edge-disjoint s-t paths
Advanced Flows	Minimum Cut	Maxflow-Mincut Theorem
	Bipartite Matchings	Vertex Cover & Independent Set
	Mincost Maxflow	Bipartite weighted matching & more

Many more (not in Algotab 2016): planarity testing, sparse matrix ordering, ...

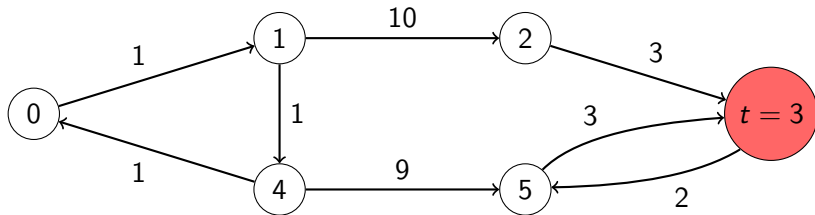
**Prerequisites:** Theory, BFS, DFS, topological sorting, Eulerian tours, Union-Find...

## Tutorial problem: statement & example

**Input** A directed graph  $G$  with positive weights on edges and a vertex  $t$ ,  
 $|V(G)| \leq 10^5$ ,  $|E(G)| \leq 2 \cdot 10^5$ .

**Definition** We call a vertex  $u$  *universal* if all vertices in  $G$  can be reached from it.

**Output** Length of a shortest path  $u \rightarrow t$  that starts in some universal vertex  $u$ .  
If such a path does not exist, output NO.



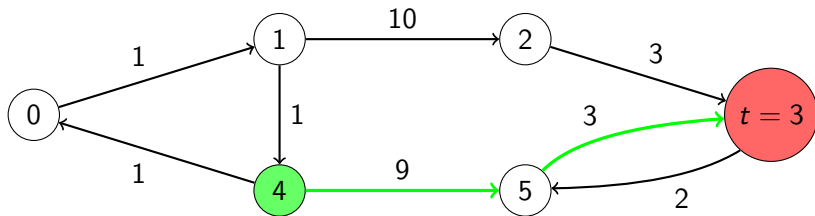


## Tutorial problem: statement & example

**Input** A directed graph  $G$  with positive weights on edges and a vertex  $t$ ,  
 $|V(G)| \leq 10^5$ ,  $|E(G)| \leq 2 \cdot 10^5$ .

**Definition** We call a vertex  $u$  *universal* if all vertices in  $G$  can be reached from it.

**Output** Length of a shortest path  $u \rightarrow t$  that starts in some universal vertex  $u$ .  
If such a path does not exist, output NO.



# Tutorial problem: how to start?

Time's short, so hurry up!

- "Check if there is a unique  $u$  with no in-edges, if yes output shortest path  $u \rightarrow t$ ."  
(**what if there is no such  $u$ ?**)
- "For each  $u$  check with DFS if  $u$  reaches all vertices, then..." (**too slow**)
- Start coding:

```
1 #include <iostream>
2 int main() {
3     // some random algorithm
4 }
```

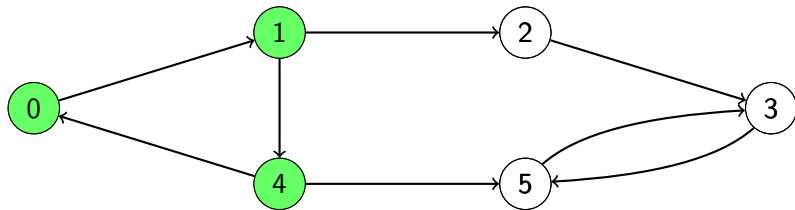
**No! Take your time,**  
model the problem,  
design the algorithm,  
**understand why it should work,**  
 $\Rightarrow$  then code.

## Tutorial problem: how to start?

- Bad question: *Why shouldn't it work?*  
("It is correct on all three examples I came up with", etc.)
- Good question: *Why should it work?*  
("How would I prove it works?")
- Applies to Moodle forums as well!

## Tutorial problem: example

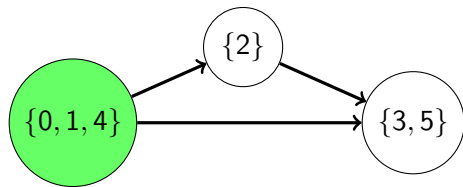
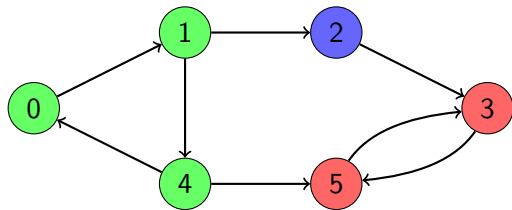
What are the universal vertices?



⇒ must be related to some sort of connected component concept in directed graphs!

## Tutorial problem: strongly connected components (SCC) example

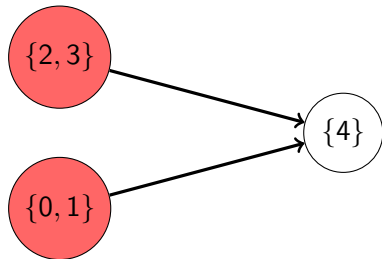
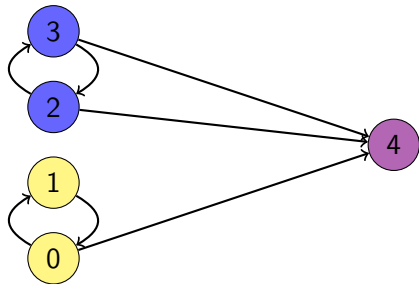
Strongly connected components:



Is there always a universal vertex?

## Tutorial problem: strongly connected components “NO” example

No!



## Tutorial problem: modeling

Let us call a strongly connected component with no in-edges in the SCC Directed Acyclic Graph a *minimal component*.

### Fact

*If there is more than one minimal component in  $G$ , then there is no universal  $u$ .*

### Lemma

*If there is exactly one minimal component in  $G$ , then its vertices are exactly the universal vertices.*

## Tutorial problem: modeling

New formulation of the problem:

- 1 If there exists  $> 1$  minimal strongly connected component in  $G$ , output NO.
- 2 Output the shortest distance  $u \rightarrow t$  for best universal  $u$  in  $G$ .

Worst-case: Still  $\Omega(|V| \cdot \text{Dijkstra's shortest paths}) = \Omega(|V|^2 \log |V| + |V||E|)$ !  
I.e. around  $|V||E| \approx 10^5 \cdot 2 \cdot 10^5 = 20000000000$  operations.



## Tutorial problem: modeling

New formulation of the problem:

- 1 If there exists  $> 1$  maximal strongly connected component in  $G_T$ , output NO.
- 2 Output the shortest distance  $t \rightarrow u$  for universal  $u$  in  $G_T$ .

Now we can work only with  $G_T$  and one single Dijkstra run!

I.e. around  $|V| \log |V| + |E| \approx 2 \cdot 10^5 = 200'000$  operations.

# Tutorial problem: implementation

First and foremost, [BGL docs](#):

- How to find the [strong\\_components](#).
- How to check how many maximal components are there?  
[topological\\_sort](#)?  
Maybe there is a simple ad hoc?
- Compute shortest  $t - u$  path on  $G_T$  with [dijkstra\\_shortest\\_paths](#).

## Tutorial problem: code – preamble

```
10 // STL includes
11 #include <iostream>
12 #include <vector>
13 #include <algorithm>
14 #include <climits>
15 // BGL includes
16 #include <boost/graph/adjacency_list.hpp>
17 #include <boost/graph/strong_components.hpp>
18 #include <boost/graph/dijkstra_shortest_paths.hpp>
19 // Namespaces
20 using namespace std;
21 using namespace boost;
```

## Tutorial problem: code – typedefs

```
24 // Directed graph with integer weights on edges.
25 typedef adjacency_list<vecS, vecS, directedS,
26     no_property,
27     property<edge_weight_t, int>
28     > Graph;
29 typedef graph_traits<Graph>::vertex_descriptor Vertex; // Vertex type
30 typedef graph_traits<Graph>::edge_descriptor Edge; // Edge type
31 typedef graph_traits<Graph>::edge_iterator EdgIt; // Edge iterator
32 // Property map edge -> weight
33 typedef property_map<Graph, edge_weight_t>::type WeightMap;
```

## Tutorial problem: code – reading the input

```
38 void testcases() {
39     // Read and build graph
40     int V, E, t;           // 1st line: <vertex_no> <edge_no> <target>
41     cin >> V >> E >> t;
42     Graph G(V);           // Creates an empty graph on V vertices
43     WeightMap weightmap = get(edge_weight, G);
44     for (int i = 0; i < E; ++i) {
45         int u, v, w;       // Each edge: <from> <to> <weight>
46         cin >> u >> v >> w;
47         Edge e; bool success; // Swapping u and v
48         tie(e, success) = add_edge(v, u, G); // to create G_T!
49         weightmap[e] = w;
50     }
```

## Tutorial problem: code – strong components

```
50 void testcases() {
51     ...
52     // Store index of the vertices' strong component; index range [0,nscc)
53     vector<int> sccmap(V); // Use this vector as exterior property map
54     int nscc = strong_components(G, // Total number of components
55         make_iterator_property_map(
56             sccmap.begin(), get(vertex_index, G)));
```

**Exterior property:** `strong_components` assigns to each vertex the index of its strong component. This is a *property* of the vertex stored *outside* of the graph itself, namely in the vector `sccmap`. To access the vector, we turn it into an *exterior property map*.

**Alternative:** Define your own *custom interior* vertex property `vertex_component_t`. See *Useful stuff: Custom properties*, page 47. Then create an (interior) property map and call the algorithm with this map (hence without `make_iterator_property_map`).

## Tutorial problem: code – maximal SCCs

```
56 void testcases() {
57     ...
58     // Find universal strong component (if any)
59     // Why does this approach work? Exercise.
60     vector<int> is_max(nsc, 1);
61     Edgelt ebegin, eend;
62     // Iterate over all edges.
63     for (tie(ebegin, eend) = edges(G); ebegin != eend; ++ebegin) {
64         // ebegin is an iterator, *ebegin is a descriptor.
65         Vertex u = source(*ebegin, G), v = target(*ebegin, G);
66         if (sccmap[u] != sccmap[v]) is_max[sccmap[u]] = 0;
67     }
68     int max_count = count(is_max.begin(), is_max.end(), true);
69     if (max_count != 1) {
70         cout << "NO" << endl;
71         return;
72     }
```

## Tutorial problem: code – Dijkstra

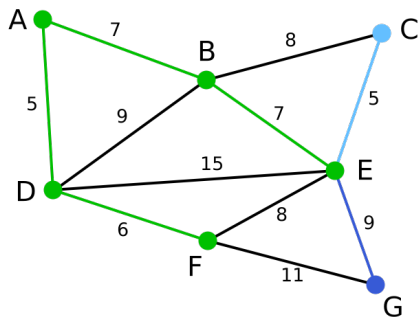
```
72 void testcases() {
73     ...
74     // Compute shortest t-u path in G_T
75     vector<int> distmap(V);    // We must use at least one of these
76     vector<Vertex> predmap(V);    // vectors as an exterior property map.
77     dijkstra_shortest_paths(G, t,
78         predecessor_map(make_iterator_property_map(    // named parameters
79             predmap.begin(), get(vertex_index, G))).
80         distance_map(make_iterator_property_map(    // concatenated by .
81             distmap.begin(), get(vertex_index, G))));
82     int res = INT_MAX;
83     for (int u = 0; u < V; ++u)
84         // Minimum of distances to 'maximal' universal vertices
85         if (is_max[sccmap[u]])
86             res = min(res, distmap[u]);
87     cout << res << endl;
88 }
```



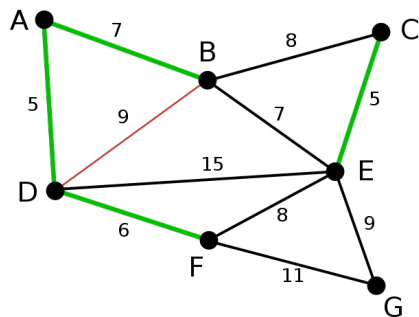
## Tutorial problem: code – main

```
93 // Main function looping over the testcases
94 int main() {
95     ios_base::sync_with_stdio(false);
96     int T;          cin >> T;    // First input line: Number of testcases.
97     while(T--)      testcases();
98     return 0;
99 }
```

# Minimum spanning trees



Intermediate step of Prim's algorithm to compute a Minimum Spanning Tree.



Intermediate step of Kruskal's algorithm to compute a Minimum Spanning Tree.

# Minimum spanning tree algorithms

Algorithm	starts with	next	Time
Prim MST	Arbitrary start vertex	Adds connection (if possible) to the closest neighbour of all so far discovered vertices.	$O(E \log V)$
Kruskal	Edge of minimum weight	Adds next smallest edge, if this is possible without creating a cycle.	$O(E \log E)$

We need to provide a predecessor vector (as an exterior property map) to Prim (maps nodes to their parents in MST), and an edge vector (to store MST edges) to Kruskal.

# Minimum spanning tree implementations

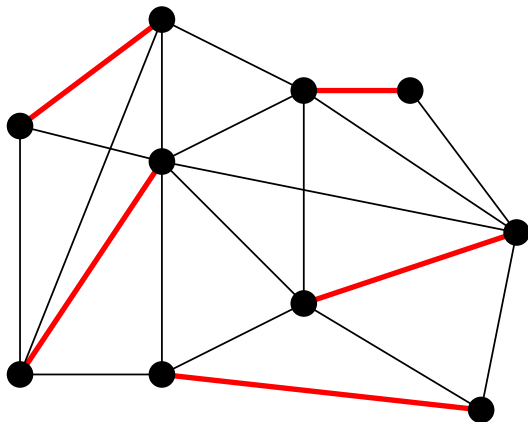
## Prim's algorithm

```
1 vector<Vertex> predmap(V);    // predecessor vector
2 Vertex start = 0;             // root vertex
3 prim_minimum_spanning_tree(G, make_iterator_property_map(
4     predmap.begin(), get(vertex_index, G)),
5     root_vertex(start));      // optional
6 for (int j = 0; j < V; ++j) {
7     Edge e; bool success;
8     tie(e, success) = edge(j, predmap[j], G);
9     if (success) {           // careful: doesn't work like this when G has loops
10         ...
```

## Kruskal's algorithm

```
1 vector<Edge> mst;             // Vector to store MST edges (not a property map!)
2 kruskal_minimum_spanning_tree(G, back_inserter(mst));
3 vector<Edge>::iterator ebegin, eend = mst.end();
4 for (ebegin = mst.begin(); ebegin != eend; ++ebegin) {
5     ...
```

## Maximum matching: general unweighted version



- $G = (V, E)$
- $M \subseteq E$  is a matching if and only if no two edges of  $M$  are adjacent.
- In an unweighted graph, a maximum matching is a matching of maximum cardinality.
- In a weighted graph, a maximum matching is a matching such that the weight sum over the included edges is maximum.
- BGL does not provide weighted matching algorithms.

## Maximum matching: invoking algorithm

```
1 // Compute Matching
2 vector<Vertex> matemap(V);
3 // Use the vector as an Exterior Property Map: Vertex -> Matched mate
4 edmonds_maximum_cardinality_matching(G, make_iterator_property_map(
5     matemap.begin(), get(vertex_index, G)));
6
7 // Look at the matching
8 // Matching size
9 int matchingsize = matching_size(G, make_iterator_property_map(
10     matemap.begin(), get(vertex_index, G)));
11
12 // unmatched vertices get the NULL_VERTEX as mate.
13 const Vertex NULL_VERTEX = graph_traits<Graph>::null_vertex();
14 for (int i = 0; i < V; ++i) {
15     if (matemap[i] != NULL_VERTEX && i < matemap[i]) {
16         ...
```

## Getting started: BGL installation

- Pre-installed in ETH computer rooms and the Algalab Virtualbox Image.  
Most likely also already installed on your system if you installed CGAL last week.
- On "standard" Linux distributions try getting a package from the repository.  
On macOS package from [Homebrew](#).
- Comments on the versions:
  - 1.58: This version is recommended (current Ubuntu LTS, Algalab VM).
  - 1.55+: These versions have Mincost-maxflow, should be fine.
  - 1.54: Prim MST bug (unless Ubuntu)

## Getting started: BGL without installing

- BGL is a Header-only library.
- Download recent version from: <http://www.boost.org/users/download/>.
- Just unpack the .tar.bz2 file, no installation required, see Section 3 here: [http://www.boost.org/doc/libs/1\\_58\\_0/more/getting\\_started/unix-variants.html](http://www.boost.org/doc/libs/1_58_0/more/getting_started/unix-variants.html).
- To build using this version of boost use this command:  
`g++ -O2 -std=c++11 -I path/to/boost_1_58_0 test.cpp -o test`
- Explanation: The '-I' flag tells the compiler to include all the files from this directory, so that it can find header files like 'boost/graph/adjacency\_list.hpp'



## Getting started: compilation problems

Error messages can be terrible.

- Consider re-compiling the code after every line after it is first written. This will help to identify the problem quickly.
- Especially after the typedefs, and again after building the graph, before you do anything else!
- There will be confusing typedefs, nested types, iterators etc. Come up with a naming pattern and stick to it.

## Getting started: runtime problems

- Isolate the smallest possible example where the program misbehaves.
- Watch out for invalidated iterators.
- Print a graph and see if it looks as expected. In particular, check if the number of vertices didn't increase due to mistakes in your edge insertion.
- More on the slides of the next (and also of the last) Section of today.

## Getting started: Using the forums

Some post	Good post
<p>I tried to solve this question as mentioned in the lecture slides, I got timelimit, I did not yet apply the</p> <p>Spoiler» sort to make it fast «, but in the slides it is mentioned that without</p> <p>Spoiler» sort «, it is still fast enough, I will be grateful if you could mention the problem with my code that makes it slow, thanks</p>	<p>My code to Problem xy gets a timelimit on the last test set and I don't know why. My approach was the following:</p> <p>Spoiler »</p> <p>«</p> <p>I can argue that my solution is correct, because</p> <p>Spoiler »</p> <p>«</p> <p>The overall running time of my solution is</p> <p>Spoiler»</p> <p>«</p> <p>Attached you can find my (reasonably commented) submission.</p>

## Getting started: Problem of the week

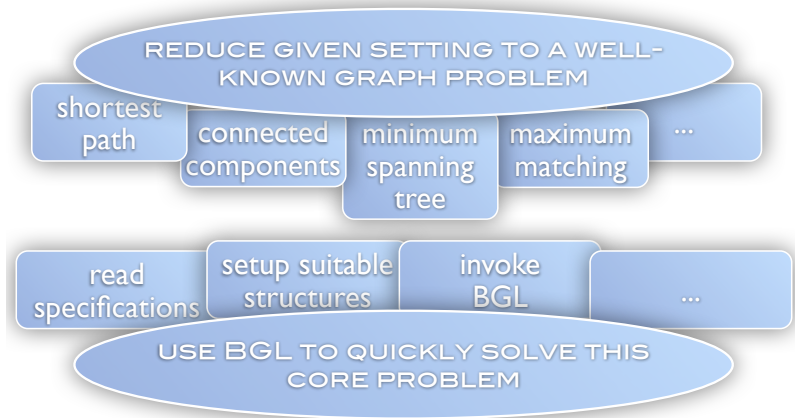
As usual, on Monday. Don't miss it!

Be advised it doesn't have to be BGL.

Anything already covered in the course can be used.



**BGL**  
THE BOOST GRAPH LIBRARY



## Useful stuff: Algolab BGL documentation

For more information please have a look at the following provided files:

**Tutorial slides** A PDF of today's tutorial. Homework: Section Useful stuff.

**Copy & paste** A PDF manual containing code snippets and some detailed explanations of the concepts presented in all BGL tutorials.

**Tutorial problem** Code and Input file of today's tutorial problem.

**Code snippets** Self contained code demonstrating many useful code snippets. Some of it can also be found in the rest of this Section.

## Useful stuff: Options for adjacency\_list

**adjacency\_list** is the class you almost always need.

```
1 // Graph Type, OutEdgeList Type, VertexList Type, (un)directedS
2 typedef adjacency_list<vecS, vecS, undirectedS,
3     no_property, // nested vertex properties
4     property<edge_weight_t, int> // nested edge properties
5     > Graph;
```

**OutEdgeList** (1st vecS) — for each vertex, adjacency list kept in a vector.  
Choosing setS instead disallows parallel edges.

**VertexList** (2nd vecS) — a list of all edges is kept in a vector. Use this!

**Directivity** directedS — directed graph.

Other choices: undirectedS (undirected graph).

Rarely needed: bidirectionalS (efficient access to incoming edges)

## Useful stuff: Building a graph

```
1 Graph G(n);      // Constructs empty graph with n vertices
2 ...
3 Edge e;
4 bool success;
5 tie(e, success) = add_edge(u, v, G);
```

- Adds edge from  $u$  to  $v$  in  $G$ .
- Caveat: if  $u$  or  $v$  don't exist in the graph,  $G$  *is automatically extended*.
- Returns an (Edge, bool) pair. First coordinate is an edge descriptor. If parallel edges are allowed, second coordinate is always true. Otherwise it is false in case of a failure (when the edge is a duplicate).



## Useful stuff: Removing vertices and edges, Clearing a graph

**Dangerous:** Deletions of single vertices and edges.

Takes time, invalidates descriptors and iterators, might behave counterintuitively.

Consult the docs. Not recommended.

```
1 remove_edge(u, v, G);
2 remove_edge(e, G);
3 clear_vertex(u, G);
4 clear_out_edges(u, G);
5 remove_vertex(u, G);
```

**OK:** Clearing a graph once it is no longer needed.

```
1 G.clear(); // Removes all edges and vertices.
2 G = Graph(n); // Destroys old graph; creates a new one with n vertices.
```

## Useful stuff: Iterators

```
1 // Iterating over vertices
2 for (u = 0; u < num_vertices(G); ++u) {
3     ...
4 // Iterating over edges
5 Edgelt eit, eend;
6 for (tie(eit, eend) = edges(G); eit != eend; ++eit) {
7     // eit is EdgeIterator, *eit is EdgeDescriptor
8     Vertex u = source(*eit, G), v = target(*eit, G);
9     ...
```

- edges(G) returns a pair of iterators which define a range of all edges.
- For undirected graphs each edge is visited once, with some orientation.

```
10 // Iterating over outgoing edges
11 OutEdgelt oeit, oeend;
12 for (tie(oeit, oeend) = out_edges(u, G); oeit != oeend; ++oeit) {
13     Vertex v = target(*oeit, G);
14     ...
```

- source(\*eit, G) is guaranteed to be u, even in an undirected graph.

## Useful stuff: Interior property maps – vertices

Think of a **property map** as a map (i.e., object with operator `[]`) indexed by vertices or edges. Property maps of vertices could be simulated with a vector, but maps of edges are very convenient.

```
1 // Note the nested syntax for defining more than one vertex property.
2 typedef adjacency_list<vecS, vecS, directedS,
3     property<vertex_name_t, string,
4         property<vertex_distance_t, int>>> Graph;
5 typedef property_map<Graph, vertex_name_t>::type NameMap;
6 typedef property_map<Graph, vertex_distance_t>::type DistMap;
7 ...
8 NameMap namemap = get(vertex_name, G);
9 namemap[u] = "Hans";
```

- `namemap` is just a handle (pointer), copying it costs  $O(1)$ .
- `vertex_name_t` is a predefined tag. It is purely conventional (you can create `property<vertex_name_t, int>` and store distances), but algorithms use them as default choices if not instructed otherwise.

## Useful stuff: Interior property maps – edges

```
1 typedef adjacency_list<vecS, vecS, directedS,
2     no_property, // No vertex properties this time.
3     // Edge properties as fifth template argument.
4     property<edge_weight_t, int>> Graph;
5 typedef property_map<Graph, edge_weight_t>::type WeightMap;
6 ...
7 WeightMap weightmap = get(edge_weight, G);
8 weightmap[e] = cost;
```

- `weightmap` is used by many algorithms (Prim, Dijkstra, Kruskal, ...)  
as default choice for the edge weight.

## Useful stuff: Predefined properties

Some *predefined* vertex and edge properties:

- `vertex_name_t`
- `vertex_distance_t`
- `vertex_color_t`
- `vertex_degree_t`
- `edge_name_t`
- `edge_weight_t`
- `edge_weight2_t`

Do not be misled into, e.g., thinking that `vertex_degree_t` will automatically keep track of the degree for you.

More: [http://www.boost.org/doc/libs/1\\_58\\_0/boost/graph/properties.hpp](http://www.boost.org/doc/libs/1_58_0/boost/graph/properties.hpp)

## Useful stuff: Custom properties

Can be defined if you want to keep additional info associated with edges.

```
1 namespace boost {
2     enum edge_info_t { edge_info = 219 }; // A unique ID.
3     BOOST_INSTALL_PROPERTY(edge, info);
4 }
5 struct EdgeInfo {
6     ...
7 };
8 ...
9 typedef adjacency_list<vecS, vecS, directedS,
10     no_property,
11     property<edge_info_t, EdgeInfo>> Graph;
12 typedef property_map<Graph, edge_info_t>::type InfoMap;
13 ...
14 InfoMap infomap = get(edge_info, G);
15 infomap[e] = ...
```

## Useful stuff: Named parameters I

Using **named parameters** is a way to pass parameters (usually property maps) to functions (BGL algorithms) which is useful in two cases:

- 1 Many algorithms have a long list of parameters. Without named parameters, all of these must be provided in the correct order, even if only some are actually needed:

```
1 // Prim non-named parameters example
2 prim_minimum_spanning_tree(G, startvertex ,
3     make_iterator_property_map(predmap.begin(), get(vertex_index, G)),
4     make_iterator_property_map(distmap.begin(), get(vertex_index, G)),
5     get(edge_weight, G), get(vertex_index, G), default_dijkstra_visitor()));
6 // Prim named parameters:
7 // PredecessorMap must be provided, all other parameters optional
8 prim_minimum_spanning_tree(G,
9     make_iterator_property_map(predmap.begin(), get(vertex_index, G)),
10    root_vertex(startvertex));
```

For e.g. Dijkstra calling the non-named parameter version is even worse!

## Useful stuff: Named parameters II

Using **named parameters** is a way to pass parameters (usually property maps) to functions (BGL algorithms) which is useful in two cases:

- 2 Some algorithms can record additional information to exterior property maps if provided by named parameters.

```
1 // Kruskal standard example
2 kruskal_minimum_spanning_tree(G, back_inserter(mst));
3 // Kruskal recording Union-Find information
4 vector<int> rankmap(num_vertices(G)); // used by Union-Find
5 vector<Vertex> predmap(num_vertices(G)); // in Union-Find, not the MST!
6 kruskal_minimum_spanning_tree(G, back_inserter(mst),
7     rank_map(make_iterator_property_map(
8         rankmap.begin(), get(vertex_index, G))). // concatenate with .
9     predecessor_map(make_iterator_property_map(
10        predmap.begin(), get(vertex_index, G))));
```

Always concatenate named parameters by a `.`

Do not pass them as separate parameters (i.e. separated by a `,`).



## Useful stuff: Where to be careful

Be careful when you deviate from the provided instructions, in particular if...

- 1 ...you use a pointer type as a property map (see e.g. [here](#)):

Buggy for Dijkstra calls in combination with Strong components header.

```
1 // What we teach (and what works):  
2 dijkstra_shortest_paths(G, 0, distance_map(  
3     make_iterator_property_map(dist.begin(), get(vertex_index, G))));  
4 // Using a pointer type (works most of the time):  
5 dijkstra_shortest_paths(G, 0, distance_map(&dist[0]));
```

- 2 ...you use [bundled properties](#) instead of nested properties:

Not well documented in the BGL examples; Buggy for MinCost flows.

- 3 ...you use named parameters for flow algorithms: Buggy for MinCost flows.  
Stick to the non-named versions, provide all property maps in correct order.