Algolab BGL Introduction

Andreas Bärtschi

October 7, 2015

BGL

Boost
Graph
Library

A **generic** C++ library of graph data structures and algorithms.
BGL docs – your new best friend:
`http://www.boost.org/doc/libs/1_57_0/libs/graph/doc`
Moodle: There's a brief **copy & paste manual**.

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 Associate *multiple* properties Accessible via *property maps*
Extensions	through function objects	through a *visitor object*, event points and methods depend on particular algorithm

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 Associate *multiple* properties Accessible via *property maps*
Extensions	through function objects	through a *visitor object*, 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	**use this!**
	Adjacency matrix	Dense graphs	*use at your*
Adaptors	Edge list	Simplicity	*own risk!*
	External adaptation	Convert existing graph structures (LEDA etc.) to BGL	Not covered in Algolab.

BGL: adjacency_list

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

```
// Easy syntax
typedef adjacency_list<vecS, vecS, directedS>    Graph;
...
// which is the same as:
typedef adjacency_list<vecS, vecS, directedS,
            no_property,
            no_property>    Graph;
...
```

BGL: adjacency_list

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

```
// Note syntax for defining more than one edge property
typedef adjacency_list<vecS, vecS, directedS,
  property<vertex_name_t, string>,  // vertex property
  property<edge_capacity_t, int,    // multiple edge properties: nested
    property<edge_residual_capacity_t, int,
      property<edge_reverse_t, Traits::edge_descriptor> > > > Graph;

typedef property_map<Graph, vertex_name_t>::type            NameMap;
typedef property_map<Graph, edge_capacity_t>::type          CapacityMap;
typedef property_map<Graph, edge_residual_capacity_t>::type ResidualMap;
typedef property_map<Graph, edge_reverse_t>::type           ReverseMap;
...
```

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

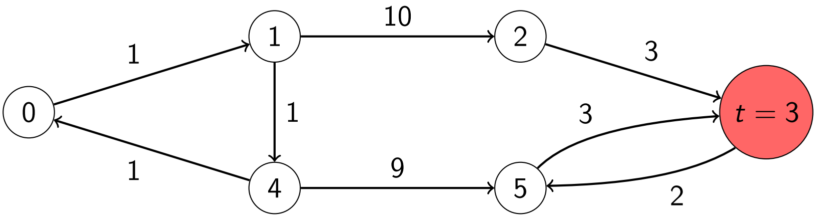
Many more (not in Algolab 2015): planarity testing, sparse matrix ordering, ...
Prerequisites: Theory, BFS, DFS, topological sorting, Eulerian tours, Union-Find...

Tutorial problem: statement

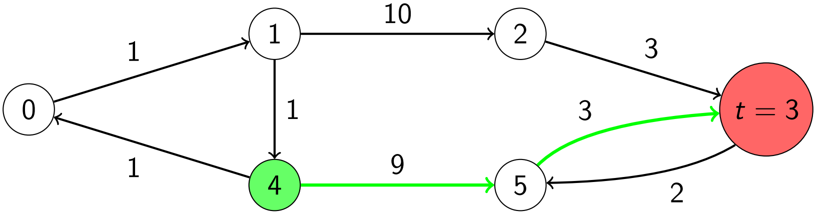
Input: A directed graph G with positive weights on edges and a vertex t.
Definition: We call a vertex u *universal* if all vertices in G can be reached from it.
Output: The length of a shortest path $u \to t$ that starts in some universal vertex u. If such a path does not exist, output `NO`.

$|V(G)| \leq 10^5, |E(G)| \leq 2 \cdot 10^5.$

Tutorial problem: example



Tutorial problem: example



Tutorial problem: how to start?

Time's short, so hurry up!

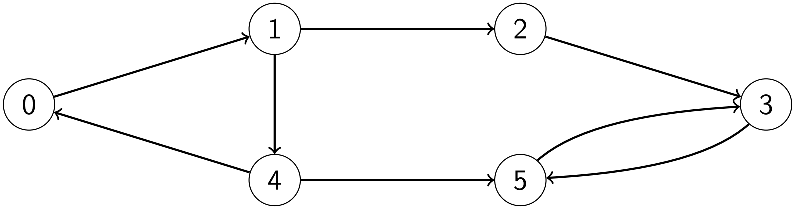
- "Check if there is a unique u with no in-edges, if yes output shortest path $u \to t$." (**what if there is no such** u**?**)
- "For each u check with DFS if u reaches all vertices, then..." (**too slow**)
- ```
  #include <iostream>
  int main()
      // some random algorithm
  ```

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

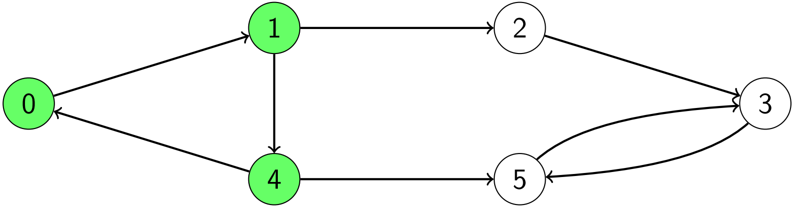
Tutorial problem: how to start?

- 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

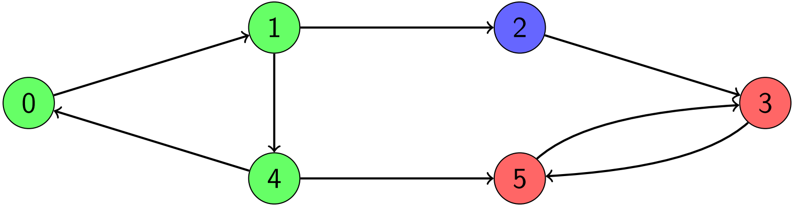


Tutorial problem: example

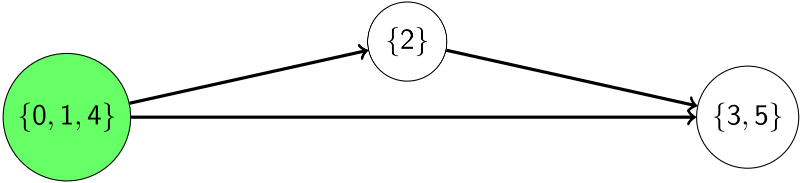


$\Rightarrow$ must be related to some sort of connected component concept in directed graphs!

Tutorial problem: strongly connected components (SCC) example



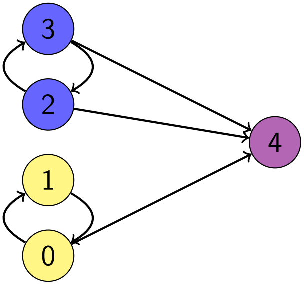
Tutorial problem: SCC directed acyclic graph example

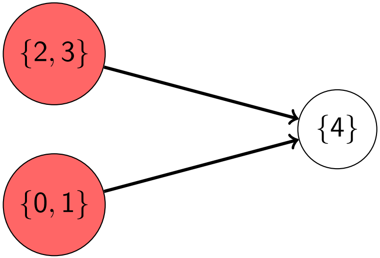


Is there always a universal vertex?

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 $u \to t$ for universal u in G.

But this is still $\Omega(n^2)$ in the worst case.

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 $t \to u$ for universal u in G_T.

Now we can work only with G_T.

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

```
#include <climits>
#include <iostream>
#include <vector>

#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/dijkstra_shortest_paths.hpp>
#include <boost/graph/strong_components.hpp>
#include <boost/tuple/tuple.hpp>  // tuples::ignore

using namespace std;
using namespace boost;
```

Tutorial problem: code – typedefs

```cpp
// Directed graph with int weights on edges.
typedef adjacency_list<vecS, vecS, directedS,
                       no_property,
                       property<edge_weight_t, int> > Graph;
typedef graph_traits<Graph>::edge_descriptor   Edge;   // Edge type
typedef graph_traits<Graph>::edge_iterator     EdgeIt; // Iterator
// Map edge -> weight.
typedef property_map<Graph, edge_weight_t>::type   WeightMap;
```

Tutorial problem: code – main

```
void testcase();

int main() {
    ios_base::sync_with_stdio(false);
    int T;  // First input line: Number of testcases.
    cin >> T;
    while (T--) testcase();
}
```

Tutorial problem: code – reading the input

```
void testcase() {
   int V, E, t; // 1st line: <ver_no> <edg_no> <tgt>
   cin >> V >> E >> t;
   Graph G(V);
   WeightMap wm = get(edge_weight, G);
   for (int i = 0; i < E; ++i) {
      int u, v, c;
      Edge e;                // Each edge: <src> <tgt> <cost>
      cin >> u >> v >> c;  // Swap u, v and instead of G
      tie(e, tuples::ignore) = add_edge(v, u, G);  // create G_T!
      wm[e] = c;
    }
```

Tutorial problem: code – strong components

```
void testcase() {
    ...
    // Store index of the vertices' strong component
    vector<int> scc(V);
    int nscc = strong_components(G,
                    make_iterator_property_map(scc.begin(),
                            get(vertex_index, G)));
```

Tutorial problem: code – maximal SCCs

```
void testcase() {
    ...
    // Find universal strong component (if any)
    // Why does this approach work? Exercise.
    vector<int> is_max(nscc, 1);
    EdgeIterator ebeg, eend;
    // Iterate over all edges.
    for (tie(ebeg, eend) = edges(G); ebeg != eend; ++ebeg) {
        // ebeg is an iterator, *ebeg is a descriptor.
        int u = source(*ebeg, G), v = target(*ebeg, G);
        if (scc[u] != scc[v]) is_max[scc[u]] = 0;
    }
```

Tutorial problem: code – maximal SCCs

```
void testcase() {
    ...
    int max_count = count(is_max.begin(), is_max.end(), true);
    if (max_count != 1) {
        cout << "NO" << endl;
        return;
    }
```

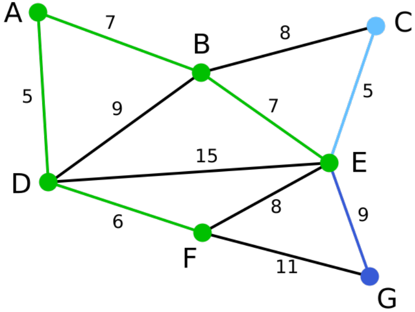
Tutorial problem: code – Dijkstra

```
void testcase() {
    ...
    //Compute shortes t-u path in G_T
    vector<int> dist(V);
    vector<int> pred(V);
    dijkstra_shortest_paths(G, t,
        predecessor_map(make_iterator_property_map(pred.begin(),
            get(vertex_index, G))).
        distance_map(make_iterator_property_map(dist.begin(),
            get(vertex_index, G))));
```

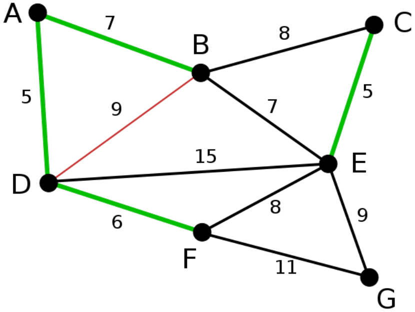
Tutorial problem: code – output

```
void testcase() {
    ...
    int res = INT_MAX;
    for (int u = 0; u < V; ++u)
        // Minimum of distances to 'maximal' vertices
        if (is_max[scc[u]])
            res = min(res, dist[u]);
    cout << res << endl;
}
```

Minimum spanning trees



Prim Minimum Spanning Tree



Kruskal 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 to Prim (stores parents in the MST), and an edge vector to Kruskal (stores the edges of the MST).

Minimum spanning tree implementations

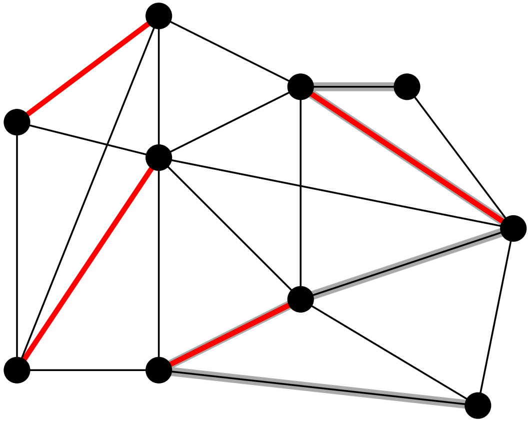
Prim's algorithm

```
vector<int> p(V);  // predecessor vector
prim_minimum_spanning_tree(G, &p[0]);
for (int j = 0; j < V; ++j) {
    Edge e; bool success;
    tie(e, success)  = edge(j, p[j], G);
    if (success) {  // careful: doesn't work like this when G has loops
        ...
```

Kruskal's algorithm

```
vector<Edge> mst;  // edge vector to store mst
kruskal_minimum_spanning_tree(G, back_inserter(mst));
vector<Edge>::iterator ebeg, eend = mst.end();
for (ebeg = mst.begin(); ebeg != eend; ++ebeg) {
   ...
```

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

```
#include <boost/graph/max_cardinality_matching.hpp>
⋮

vector<VertexDescriptor> mate(n);
edmonds_maximum_cardinality_matching(g, &mate[0]);

const VertexDescriptor NULL_VERTEX = graph_traits<Graph>::null_vertex();
...
for(int i = 0; i < n; i++)
   if(mate[i] != NULL_VERTEX && i < mate[i])
     cout << i << " - " << mate[i] << endl;
```

Getting started: BGL installation

- Pre-installed in ETH computer rooms and the Algolab 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 MacPorts.
- Comments on the versions:
  1.57: This version runs on the judge.
  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_57_0/more/getting_started/unix-variants.html`.
- To build using this version of boost use this command:
  g++ -I path/to/boost_1_57_0 example.cpp -o example
- Explanation: The '-I' flag tells the compile 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 last) Section of today.

Getting started: Using the forums

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

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.

Conclusion

boost
C++ LIBRARIES

BGL
THE BOOST GRAPH LIBRARY

REDUCE GIVEN SETTING TO A WELL-KNOWN GRAPH PROBLEM

- shortest path
- connected components
- minimum spanning tree
- maximum matching
- ...

- read specifications
- setup suitable structures
- invoke BGL
- ...

USE BGL TO QUICKLY SOLVE THIS CORE PROBLEM

Useful stuff: Namespace

Simple: everything is in namespace `boost`.

```
using namespace boost;
```

Useful stuff: Tuple utility functions

A handy trick when function returns a tuple:

```
#include <boost/tuple/tuple.hpp>
...
int a;
double b;
tie(a, b) = make_pair(1, 0.1);
tie(tuples::ignore, b) = make_pair(2, -0.3);
```

Useful stuff: adjacency_list

This is the class you almost always need.

```
#include <boost/graph/adjacency_list.hpp>
...
typedef adjacency_list<vecS, vecS, directedS> Graph;
```

- 1st `vecS` — for each vertex, adjacency list kept in a `vector`.
  Choosing `setS` instead disallows parallel edges.
- 2nd `vecS` — a list of all edges is kept in a `vector`.
- `directedS` — directed graph.
  Other choice: `undirectedS`.

Useful stuff: Vertices and edges

- **Vertex descriptor**: This is (almost) always an `int` in range `[0, num_vertices(G))`. Don't overcomplicate this.
- **Edge descriptor**: an object that represents a single edge.

```
typedef graph_traits<Graph>::edge_descriptor Edge;
Edge e;
int u = source(e, G), v = target(e, G);
```

Useful stuff: Building a graph

```
Graph G(n);  // Constructs empty graph with n vertices
...
Edge e;
bool ok;
tie(e, ok) = 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 failure when the edge is a duplicate.

Useful stuff: Removing vertices and edges (dangerous!)

```
remove_edge(u, v, G);
remove_edge(e, G);
clear_vertex(u, G);
clear_out_edges(u, G);
remove_vertex(u, G);
```

- Consult the docs. Takes time, invalidates descriptors and iterators, might behave counterintuitively. Not recommended.

Useful stuff: Clearing a graph

```
G.clear();
```

- Removes all edges and vertices.

```
G = Graph(n);
```

- Destroys the old graph and creates a new one with `n` vertices.

Useful stuff: Iterating over vertices and edges

```
// Iterating over vertices
for (int u = 0; u < num_vertices(G); ++u) {
  ...
}
// Iterating over edges
typedef graph_traits<Graph>::edge_iterator EdgeIterator;
EdgeIterator eit, eend;
for (tie(eit, eend) = edges(G); eit != eend; ++eit) {
   // eit is EdgeIterator, *eit is EdgeDescriptor
   int u = source(*eit, G), v = target(*eit, G);
   ...
}
```

- `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.

Useful stuff: Iterating over edges incident to a vertex

```
// Iterating over outgoing edges
typedef graph_traits<Graph>::out_edge_iterator
   OutEdgeIterator;
OutEdgeIterator eit, eend;
for (tie(eit, eend) = out_edges(u, G); eit != eend; ++eit) {
   int v = target(*eit, G);
   ...
}
```

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

Useful stuff: Property maps

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

Useful stuff: Vertex property map

```
// Note syntax for defining more than one map.
typedef adjacency_list<vecS, vecS, directedS,
   property<vertex_name_t, string,
      property<vertex_distance_t, int> > > Graph;
typedef property_map<Graph, vertex_name_t>::type NameMap;
...
NameMap name_map = get(vertex_name, G);
name_map[u] = "Hans";
```

- `name_map` 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: Edge property map

```
typedef adjacency_list<vecS, vecS, directedS,
   no_property,  // Vertex properties, none this time.
   // Edge properties as fifth template argument.
   property<edge_weight_t, int> > > Graph;
typedef property_map<Graph, edge_weight_t>::type
   WeightMap;
...
EdgeDescriptor e;
...
WeightMap wm = get(edge_weight, G);
wm[e] = k;
```

- To close nested templates > > must be used instead of >>.

Useful stuff: Some predefined properties

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

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

Useful stuff: Defining a custom property

Convenient e.g., if you want to keep additional info associated with edges.

```
namespace boost {
   enum edge_info_t { edge_info = 219 };  // A unique ID.
   BOOST_INSTALL_PROPERTY(edge, info);
}
struct EdgeInfo {
   ...
};
```

Useful stuff: Defining a custom property (cont.)

```
typedef adjacency_list<vecS, vecS, directedS,
   no_property,
   property<edge_info_t, EdgeInfo> > Graph;
typedef property_map<Graph, edge_info_t>::type InfoMap;
...
InfoMap im = get(edge_info, G);
im[e] = ...
```

Useful stuff: Calling an algorithm in different ways

- Example: `kruskal_minimum_spanning_tree`.
- Follow the doc page.
- Header: `#include <boost/graph/kruskal_min_spanning_tree.hpp>`

```
vector<Edge> mst;          // edge list to store mst
kruskal_minimum_spanning_tree(G, back_inserter(mst));
vector<Edge>::iterator ebeg, eend = mst.end();
// Go through minimum spanning tree
for (ebeg = mst.begin(); ebeg != eend; ++ebeg) {
   ...
}
```

- `edge_weight_t` map must be defined for this to work.

Useful stuff: Calling an algorithm in different ways – named parameters

- Maybe you might want to access additional information computed by the Union-Find algorithm of Kruskal. For example you can access `rank_map` and `predecessor_map`.
- You need to provide additional custom arguments. This is done via *named parameters*.

```
vector<int> R(num_vertices(G)), P(num_vertices(G));
kruskal_minimum_spanning_tree(G, back_inserter(mst),
   weight_map(get(edge_weight, G)).
      rank_map(&R[0]).  // A dot, not a comma!
      predecessor_map(&P[0]));
```

- To pass a vector as a vertex property map you need to provide `&V[0]`, an iterator like `V.begin()` will not work.
- Defaults: `get(edge_weight, G)` for `weight_map`, internally created vectors for `rank_map` and `predecessor_map`.