Algolab BGL Introduction

Andreas Bärtschi

October 12, 2016

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_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 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 the nested 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,              // nested edge properties
        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 2016): planarity testing, sparse matrix ordering, ...

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

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 2016): planarity testing, sparse matrix ordering, ...

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

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 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.

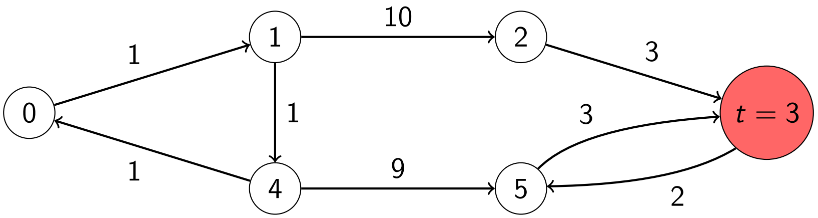
Output Length of a shortest path $u \to t$ that starts in some universal vertex u. If such a path does not exist, output `NO`.

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 $u \to t$ that starts in some universal vertex u. If such a path does not exist, output `NO`.

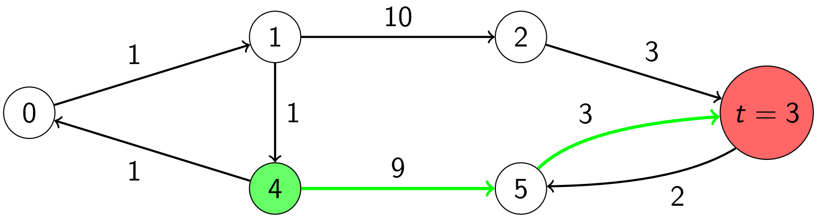


Tutorial problem: statement & example

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 $u \to t$ 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 $u \to t$." (**what if there is no such u?**)
- "For each u check with DFS if u reaches all vertices, then..." (**too slow**)
- Start coding:

```
#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?

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**)
- Start coding:

```
#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?

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**)
- Start coding:

```
#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?

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**)
- Start coding:

```
#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?

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**)
- Start coding:

```
#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.

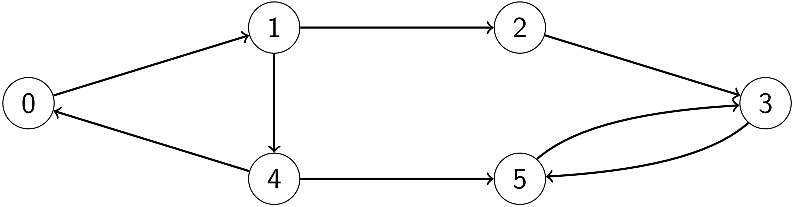
Tutorial problem: how to start?

- 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!

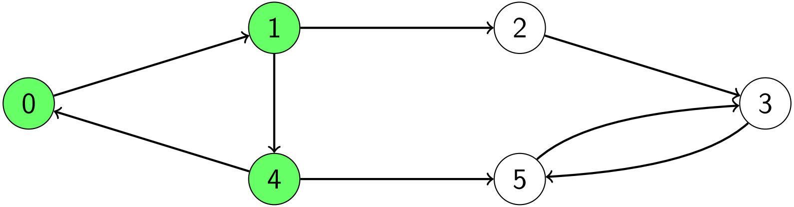
Tutorial problem: example

What are the universal vertices?



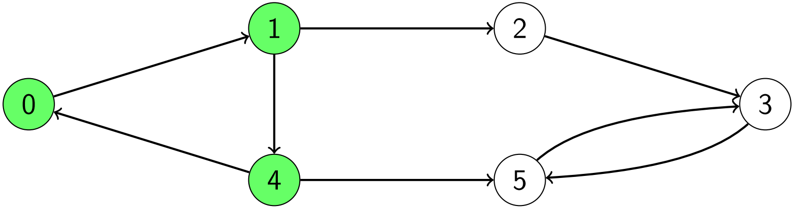
Tutorial problem: example

What are the universal vertices?



Tutorial problem: example

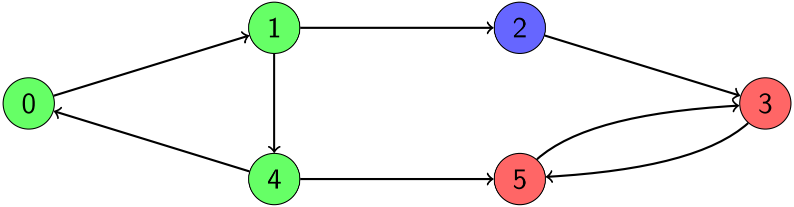
What are the universal vertices?



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

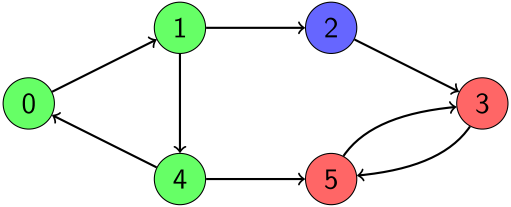
Tutorial problem: strongly connected components (SCC) example

Strongly connected components:



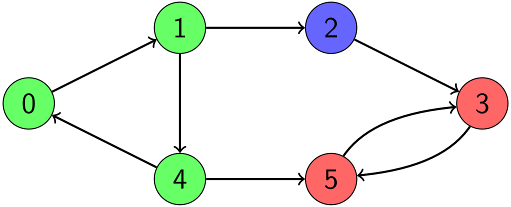
Tutorial problem: strongly connected components (SCC) example

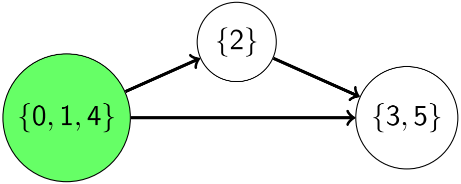
Strongly connected components:



Tutorial problem: strongly connected components (SCC) example

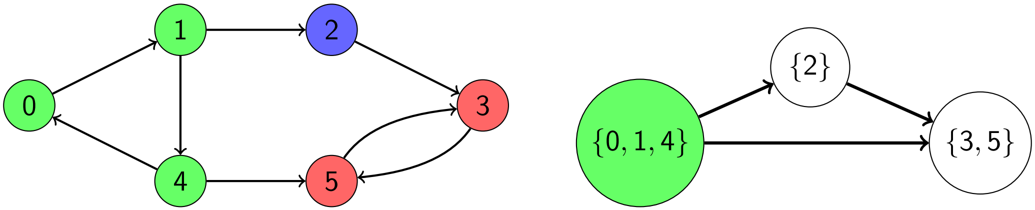
Strongly connected components:





Tutorial problem: strongly connected components (SCC) example

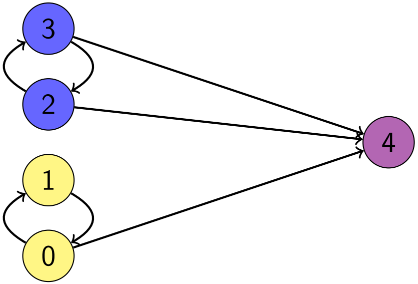
Strongly connected components:

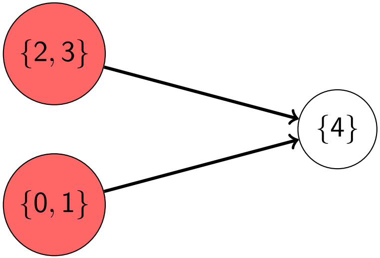


Is there always a universal vertex?

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

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 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 minimal strongly connected component in G, output `NO`.
2. Output the shortest distance $u \to t$ 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 $t \to u$ 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: 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: 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: 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

```
// STL includes
#include <iostream>
#include <vector>
#include <algorithm>
#include <climits>
// BGL includes
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/strong_components.hpp>
#include <boost/graph/dijkstra_shortest_paths.hpp>
// Namespaces
using namespace std;
using namespace boost;
```

Tutorial problem: code – typedefs

```
// Directed graph with integer weights on edges.
typedef adjacency_list<vecS, vecS, directedS,
    no_property,
    property<edge_weight_t, int>
   >                                             Graph;
typedef graph_traits<Graph>::vertex_descriptor Vertex; // Vertex type
typedef graph_traits<Graph>::edge_descriptor   Edge;   // Edge type
typedef graph_traits<Graph>::edge_iterator     EdgeIt; // Edge iterator
// Property map edge -> weight
typedef property_map<Graph, edge_weight_t>::type    WeightMap;
```

Tutorial problem: code – reading the input

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

Tutorial problem: code – strong components

```
void testcases() {
    ...
    // Store index of the vertices' strong component
    vector<Vertex> sccmap(V); // Use this vector as exterior property map
    int nscc = strong_components(G,
            make_iterator_property_map(
                sccmap.begin(), get(vertex_index, G)));
```

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.

Tutorial problem: code – maximal SCCs

```cpp
void testcases() {
    ...
    // Find universal strong component (if any)
    // Why does this approach work? Exercise.
    vector<int> is_max(nscc, 1);
    EdgeIt ebeg, eend;
    // Iterate over all edges.
    for (tie(ebeg, eend) = edges(G); ebeg != eend; ++ebeg) {
        // ebeg is an iterator, *ebeg is a descriptor.
        Vertex u = source(*ebeg, G), v = target(*ebeg, G);
        if (sccmap[u] != sccmap[v])     is_max[sccmap[u]] = 0;
    }
    int max_count = count(is_max.begin(), is_max.end(), true);
    if (max_count != 1) {
        cout << "NO" << endl;
        return;
    }
```

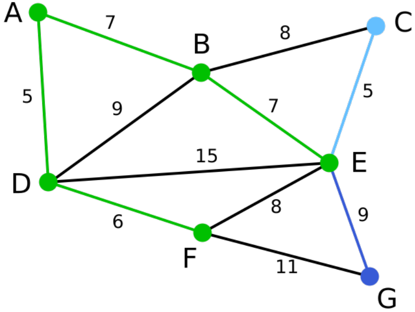
Tutorial problem: code – Dijkstra

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

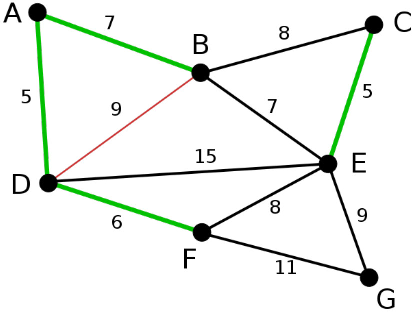
Tutorial problem: code – main

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

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.	$O(E \log E)$

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

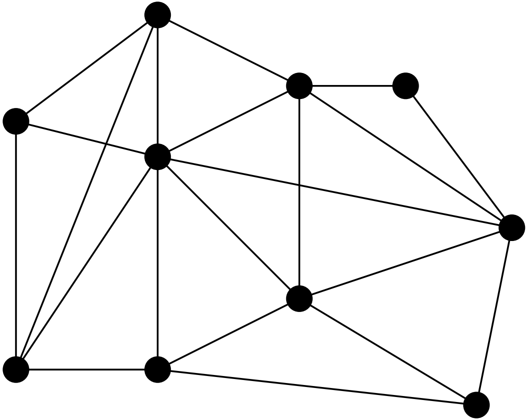
Prim's algorithm

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

Kruskal's algorithm

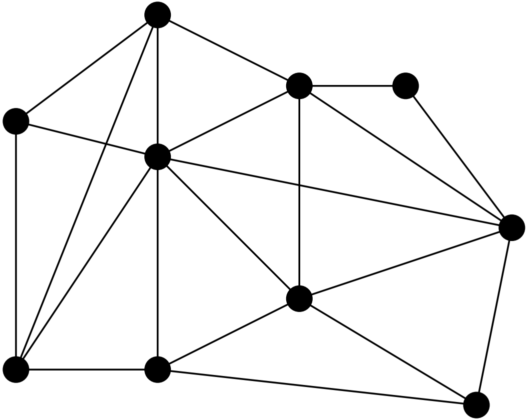
```
vector<Edge> mst;     // Vector to store MST edges (not a property map!)
kruskal_minimum_spanning_tree(G, back_inserter(mst));
vector<Edge>::iterator ebeg, eend = mst.end();
for (ebeg = mst.begin(); ebeg != eend; ++ebeg) {
    ...
```

Maximum matching: general unweighted version



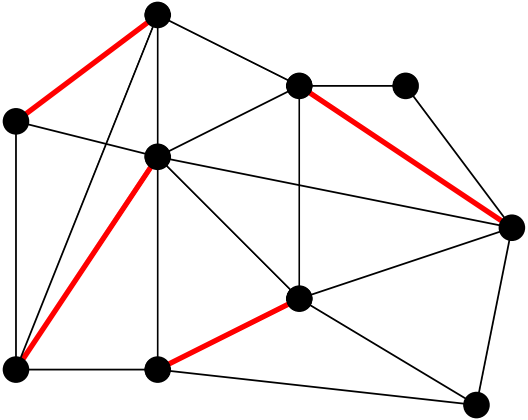
- $G = (V, E)$
- $M \subseteq E$ is a matching if and only if no two edges of M are adjacent.
- In an unweighted graph, a maximum matching is a matching of maximum cardinality.
- In a weighted graph, a maximum matching is a matching such that the weight sum over the included edges is maximum.
- BGL does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

Maximum matching: general unweighted version



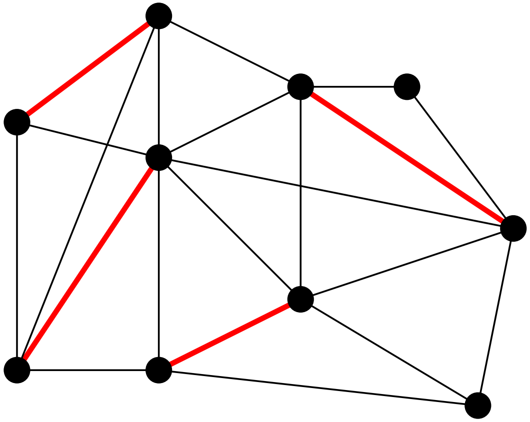
- $G = (V, E)$
- $M \subseteq E$ is a matching if and only if no two edges of M are adjacent.
- In an unweighted graph, a maximum matching is a matching of maximum cardinality.
- In a weighted graph, a maximum matching is a matching such that the weight sum over the included edges is maximum.
- BGL does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

Maximum matching: general unweighted version



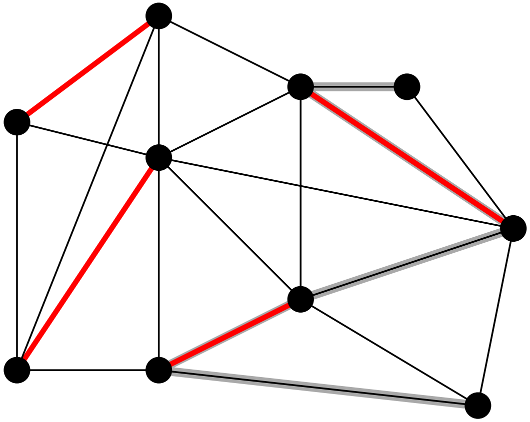
- $G = (V, E)$
- $M \subseteq E$ is a matching if and only if no two edges of M are adjacent.
- In an unweighted graph, a maximum matching is a matching of maximum cardinality.
- In a weighted graph, a maximum matching is a matching such that the weight sum over the included edges is maximum.
- BGL does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

Maximum matching: general unweighted version



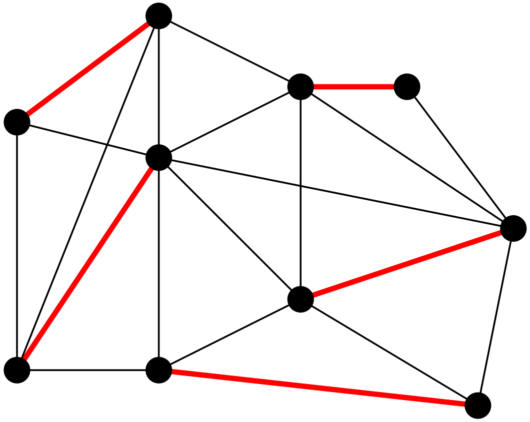
- $G = (V, E)$
- $M \subseteq E$ is a matching if and only if no two edges of M are adjacent.
- In an unweighted graph, a maximum matching is a matching of maximum cardinality.
- In a weighted graph, a maximum matching is a matching such that the weight sum over the included edges is maximum.
- BGL does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

Maximum matching: general unweighted version



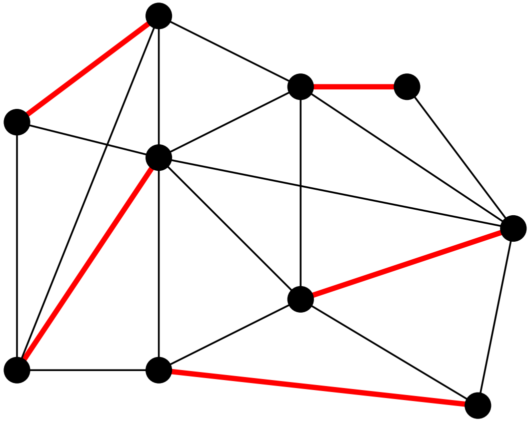
- $G = (V, E)$
- $M \subseteq E$ is a matching if and only if no two edges of M are adjacent.
- In an unweighted graph, a maximum matching is a matching of maximum cardinality.
- In a weighted graph, a maximum matching is a matching such that the weight sum over the included edges is maximum.
- BGL does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

Maximum matching: general unweighted version



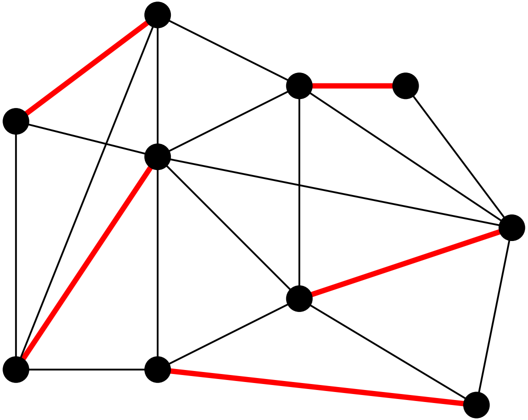
- $G = (V, E)$
- $M \subseteq E$ is a matching if and only if no two edges of M are adjacent.
- In an unweighted graph, a maximum matching is a matching of maximum cardinality.
- In a weighted graph, a maximum matching is a matching such that the weight sum over the included edges is maximum.
- BGL does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

Maximum matching: 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: 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

```
// Compute Matching
vector<Vertex> matemap(V); // Use as an Exterior Property Map: Vertex -> Matc
edmonds_maximum_cardinality_matching(G, make_iterator_property_map(
        matemap.begin(), get(vertex_index, G)));

// Look at the matching
// Matching size
int matchingsize = matching_size(G, make_iterator_property_map(
        matemap.begin(), get(vertex_index, G)));

// unmatched vertices get the NULL_VERTEX as mate.
const Vertex NULL_VERTEX = graph_traits<Graph>::null_vertex();
for (int i = 0; i < V; ++i) {
    if (matemap[i] != NULL_VERTEX && i < matemap[i]) {
        ...
```

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 Homebrew.
- Comments on the versions:
  1.58: This version is recommended (current Ubuntu LTS, Algolab VM).
  1.55+: These versions have Mincost-maxflow, should be fine.
  1.54: Prim MST bug (unless Ubuntu)

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 Homebrew.
- Comments on the versions:
  1.58: This version is recommended (current Ubuntu LTS, Algolab VM).
  1.55+: These versions have Mincost-maxflow, should be fine.
  1.54: Prim MST bug (unless Ubuntu)

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 Homebrew.
- Comments on the versions:
  1.58: This version is recommended (current Ubuntu LTS, Algolab 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`.
- 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 compile to include all the files from this directory, so that it can find header files like 'boost/graph/adjacency_list.hpp'

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`.
- 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 compile to include all the files from this directory, so that it can find header files like 'boost/graph/adjacency_list.hpp'

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`.
- 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 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: 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: 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: 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: 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
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.

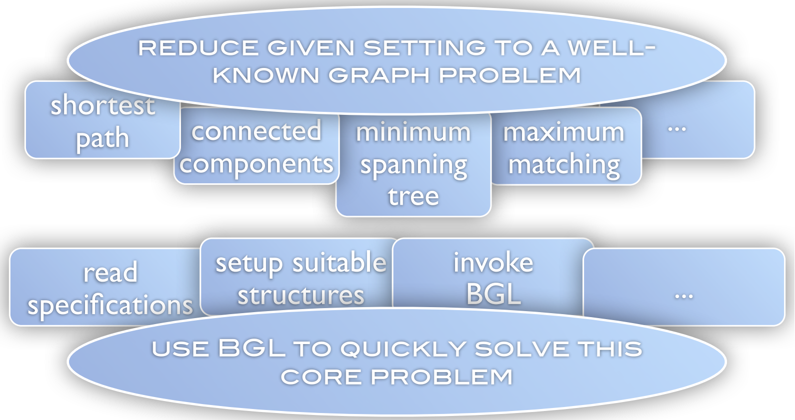
Getting started: Problem of the week

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



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.

```
// Graph Type, OutEdgeList Type, VertexList Type, (un)directedS
typedef adjacency_list<vecS, vecS, undirectedS,
    no_property,                       // nested vertex properties
    property<edge_weight_t, int>       // nested edge properties
    >                                  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: Options for adjacency_list

adjacency_list is the class you almost always need.

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

```
Graph G(n);     // Constructs empty graph with n vertices
...
Edge e;
bool success;
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: Building a graph

```
Graph G(n);     // Constructs empty graph with n vertices
...
Edge e;
bool success;
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: Building a graph

```
Graph G(n);    // Constructs empty graph with n vertices
...
Edge e;
bool success;
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.

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

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

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

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.

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

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

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

Useful stuff: Iterators

```
// Iterating over vertices
for (u = 0; u < num_vertices(G); ++u) {
    ...
// Iterating over edges
EdgeIt eit, eend;
for (tie(eit, eend) = edges(G); eit != eend; ++eit) {
    // eit is EdgeIterator, *eit is EdgeDescriptor}
    Vertex 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.

```
// Iterating over outgoing edges
OutEdgeIt oeit, oeend;
for (tie(oeit, oeend) = out_edges(u, G); oeit != oeend; ++oeit) {
    Vertex v = target(*oeit, G);
    ...
```

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

Useful stuff: Iterators

```
// Iterating over vertices
for (u = 0; u < num_vertices(G); ++u) {
    ...
// Iterating over edges
EdgeIt eit, eend;
for (tie(eit, eend) = edges(G); eit != eend; ++eit) {
    // eit is EdgeIterator, *eit is EdgeDescriptor}
    Vertex 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.

```
// Iterating over outgoing edges
OutEdgeIt oeit, oeend;
for (tie(oeit, oeend) = out_edges(u, G); oeit != oeend; ++oeit) {
    Vertex v = target(*oeit, G);
    ...
```

- `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.

```
// Note the nested syntax for defining more than one vertex property.
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;
typedef property_map<Graph, vertex_distance_t>::type DistMap;
...
NameMap namemap = get(vertex_name, G);
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 – 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.

```
// Note the nested syntax for defining more than one vertex property.
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;
typedef property_map<Graph, vertex_distance_t>::type  DistMap;
...
NameMap namemap = get(vertex_name, G);
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

```
typedef adjacency_list<vecS, vecS, directedS,
    no_property, // No vertex properties this time.
    // Edge properties as fifth template argument.
    property<edge_weight_t, int> >                Graph;
typedef property_map<Graph, edge_weight_t>::type    WeightMap;
...
WeightMap weightmap = get(edge_weight, G);
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`

Useful stuff: Custom properties

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