Algolab Graph and BGL Introduction

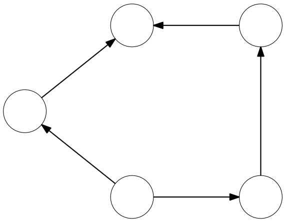
Chih-Hung Liu, slides from Daniel Wolleb, Petar Ivanov, Andreas Bärtschi

October 10, 2018

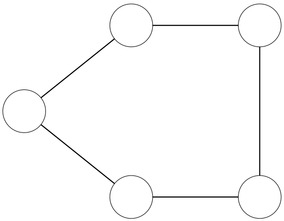
Graph definitions

A graph has n vertices/nodes V and m edges/arcs E.

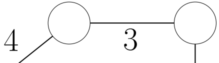
- **(un-) directed**
- **(non-) weighted**
- **(a-) cyclic**
- **(dis-) connected**

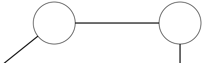


Directed



Undirected





Complexity-driven programming (not yet a real thing):

- $\Theta(V+E)$ – great! $E < 10^{7\dots9}$
- $\Theta(V \cdot \log(V+E))$ – cool
- $\Theta(V \cdot E)$ – maybe ok
- $\Theta(2^V)$ – slow, $V < 20\dots40$

General note:

! approach$^{\text{Find shortest paths}}$ $\neq$ algorithm$^{\text{Dijkstra}}$ $\neq$ implementation$^{\text{with adj.matrix}}$

! abstract data type$^{\text{Dictionary}}$ $\neq$ data structure$^{\text{Red-Black tree}}$ $\neq$ implementation$^{\text{as in STL}}$

BGL

Boost
Graph
Library

A **generic** C++ library of graph data structures and algorithms.
BGL docs – your new best friend:
`https://www.boost.org/doc/libs/1_68_0/libs/graph/doc/index.html`
and also on the judge:
`https://judge.inf.ethz.ch/doc/algolab/bgl/boost_1_66_0/libs/graph`
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 (not covered in Algolab)	through function objects	through a *visitor object*, event points and methods depend on particular algorithm

BGL: Graph Representations

Representation	Advantages	Do
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 own risk!*

BGL: adjacency_list

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

```
// Easy syntax. Parameters:
// OutEdgeList type, VertexList type, Directivity
typedef boost::adjacency_list<boost::vecS, boost::vecS, boost::directedS>    Graph;

// which is the same as:
typedef boost::adjacency_list<boost::vecS, boost::vecS, boost::directedS,
boost::no_property,    // the graph has no interior vertex properties
boost::no_property     // the graph has no interior edge properties
>               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 44*.)

BGL: adjacency_list

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

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

typedef boost::property_map<Graph, boost::vertex_name_t>::type            NameMap;
typedef boost::property_map<Graph, boost::edge_capacity_t>::type          CapacityMap;
typedef boost::property_map<Graph, boost::edge_residual_capacity_t>::type ResidualMap;
typedef boost::property_map<Graph, boost::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 mapping (object with `operator []`). Also see *Useful stuff: Interior property maps, pages 48–49*.

Warm-up: Read a Graph

```cpp
typedef boost::adjacency_list<vecS, vecS, directedS> Graph;

int main()
{
    Graph G(4);

    boost::add_edge(0, 1, G);
    boost::add_edge(1, 2, G);
    boost::add_edge(2, 3, G);
    boost::add_edge(3, 0, G);

    boost::graph_traits<Graph>::edge_iterator ebeg, eend;
    for(boost::tie(ebeg, eend) = boost::edges(G); ebeg != eend; ++ebeg){
        std::cout << "G contains an edge from " << boost::source(*ebeg, G)
                  << "to "  << boost::target(*ebeg, G) << std::endl;
    }
}
```

Warm-up: Read a weighted Graph

```
typedef boost::adjacency_list<vecS, vecS, directedS, no_property,
                                    property<edge_weight_t, int> > Graph;
typedef boost::property_map<Graph, edge_weight_t>::type WeightMap;
typedef Graph::edge_descriptor Edge;


int main()
{
Graph G(4);
WeightMap weights = boost::get(edge_weight, G);
bool added;
Edge e;
boost::tie(e, added) = boost::add_edge(0, 1, G); weights[e]=1;
boost::tie(e, added) = boost::add_edge(1, 2, G); weights[e]=3;
boost::tie(e, added) = boost::add_edge(2, 3, G); weights[e]=-2;
boost::tie(e, added) = boost::add_edge(3, 0, G); weights[e]=5;

graph_traits<Graph>::edge_iterator ebeg, eend;
for(boost::tie(ebeg, eend) = boost::edges(G); ebeg != eend; ++ebeg)
    std::cout << "Edge (" << boost::source(*ebeg G) << ", "  << boost::target(*ebeg, G)
                  << ") has weight " << weights[*ebeg] << std::endl;
}
```

BGL: Graph Algorithms

Area	Topic	Details
Basics	Distances	Dijkstra shortest paths
		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 2018): planarity testing, sparse matrix ordering, ...
Prerequisites: Graph theory, BFS, DFS, topological sorting, Eulerian tours, Union-Find...

Recap: Graph Traversal and Shortest paths

Graph Traversal with vertices partitioned into three sets: visited, enqueued, unknown

- **BFS** – closest first, $\mathcal{O}(V + E)$
- **DFS** – furthest first, $\mathcal{O}(V + E)$
- **Dijkstra** – weighted closest first, $\mathcal{O}(E + V \log V)$ (BGL docs)

Shortest paths with negative weights: Induction on number of edges in subpaths

- **Bellman Ford** – shortest paths from a single source, $\mathcal{O}(V \cdot E)$ (BGL docs)
- **Floyd–Warshall** – all-pairs shortest paths, $\mathcal{O}(V^3)$ (BGL docs)

Both can also be used to detect negative cycles

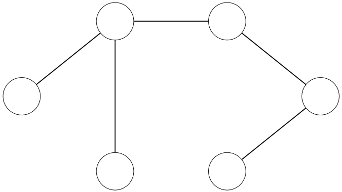
Recap: Components (Vertex Connectivity)

Undirected Graph:

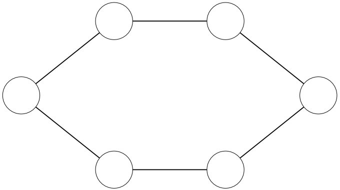
- **Connected** – vertex pair with a path
- **Biconnected** – vertex pair in a cycle
- **Articulation points** – vertex disconnecting a component
- **Bridges** – edge disconnecting a component

Directed Graph:

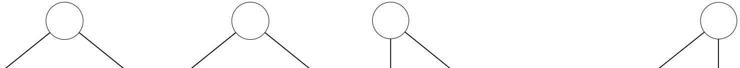
- **Strongly connected** – vertex pair with directed path in both ways



Connected



Bi-connected

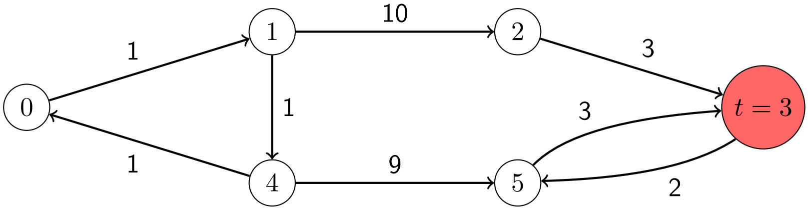


Tutorial problem: statement & example

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 Minimum length of a shortest path $u \to t$ over all universal vertices u. If such a path does not exist, output `NO`.

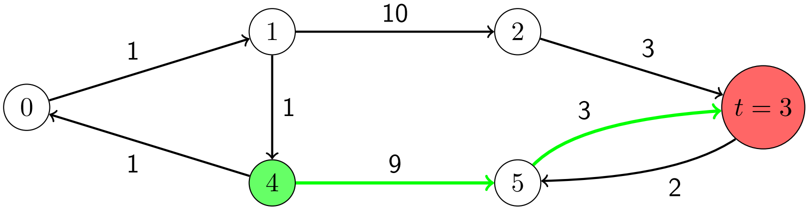


Tutorial problem: statement & example

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 Minimum length of a shortest path $u \to t$ over all universal vertices 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
}
```

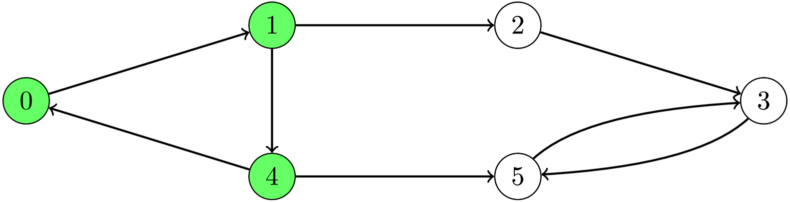
No! **Take your time**,
model the problem,
design the algorithm,
understand why it should work,
$\Rightarrow$ then code.

Tutorial problem: how to start?

- 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?")

Tutorial problem: example

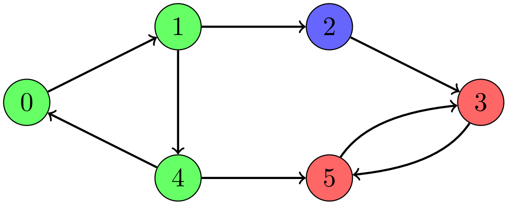
What are the universal vertices?

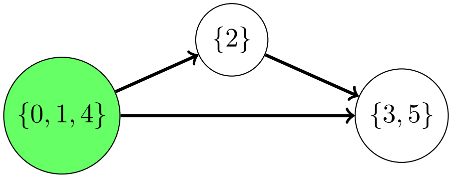


$\Rightarrow$ must be related to some sort of connected component concept in directed graphs!

Tutorial problem: strongly connected components (SCC) example

Strongly connected components:

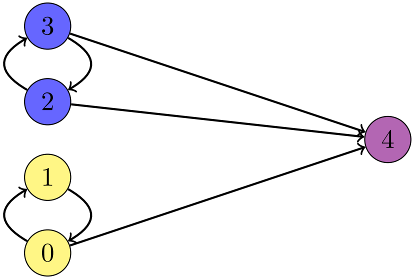


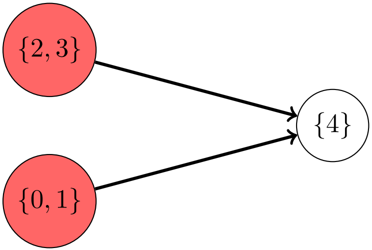


Is there always a universal vertex?

Tutorial problem: strongly connected components “NO” example

No!





Tutorial problem: modeling

Let us call a strongly connected component a *minimal component* if it has no in-edges in the strong condensation of the graph (the directed acyclic graph of the strongly connected components).

Fact

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

Step 1: $O(|V| + |E|)$ time (DFS)

Step 2: Dijkstra's algorithm $\Omega(|V|)$ times $\Rightarrow \Omega(|V|^2 \log |V| + |V||E|)$ time!
I.e. around $|V||E| \approx 10^5 \cdot 2 \cdot 10^5 = 2\,\underbrace{0'000'000'000}_{\text{too many zeros}}$ operations.

Tutorial problem: modeling

Another new formulation of the problem:

1. We work with the reversed graph G_T, where all the edges of G are reversed.
2. If there exists > 1 maximal strongly connected component in G_T, output `NO`.
   (maximal component: no out-edge & minimal component: no in-edge)
3. Output the shortest distance $t \to u$ for any vertex u in the unique maximal strongly connected component of 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

How to implement this now?
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 solution?
- 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 <limits>
// BGL includes
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/strong_components.hpp>
#include <boost/graph/dijkstra_shortest_paths.hpp>
```

Tutorial problem: code – typedefs

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

Tutorial problem: code – reading the input

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

Tutorial problem: code – strong components

```cpp
void testcase() {
...
// Store index of the vertices' strong component; index range [0,nscc)
std::vector<int> sccmap(V);  // Use this vector as exterior property map
int nscc = boost::strong_components(GT,  // Total number of components
boost::make_iterator_property_map(
sccmap.begin(), boost::get(boost::vertex_index, GT)));
```

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*, i.e., using *boost::make_iterator_property_map*.

Tutorial problem: code – maximal SCCs

```cpp
void testcase() {
...
// Find universal strong component (if any)
// Why does this approach work? Exercise.
std::vector<bool> is_max(nscc, true);
EdgeIt ebeg, eend;
// Iterate over all edges.
for (boost::tie(ebeg, eend) = boost::edges(GT); ebeg != eend; ++ebeg) {
// ebeg is an iterator, *ebeg is a descriptor.
    Vertex u = boost::source(*ebeg, GT), v = boost::target(*ebeg, GT);
    if (sccmap[u] != sccmap[v])    is_max[sccmap[u]] = false;
    // this edge (u,v) in GT implies that component sccmap[u] is not minimal in G
}
int max_count = std::count(is_max.begin(), is_max.end(), true);
if (max_count != 1) {
    std::cout << "NO" << std::endl;
return;
}
```

Tutorial problem: code – Dijkstra

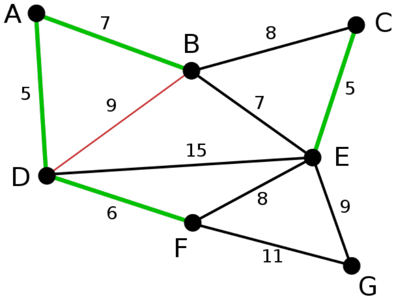
```cpp
void testcase() {
...
// Compute shortest t-u path in GT
std::vector<int> distmap(V);    // We must use at least one of these
std::vector<Vertex> predmap(V);    // vectors as an exterior property map.
boost::dijkstra_shortest_paths(GT, t,
predecessor_map(boost::make_iterator_property_map(  //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

Tutorial problem: code – main

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

Recap: Minimum spanning trees

For a connected graph $G = (V, E)$, a mininum spanning tree of G is a subgraph of G connecting all vertices in V with the minimum sum of edge weights.



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

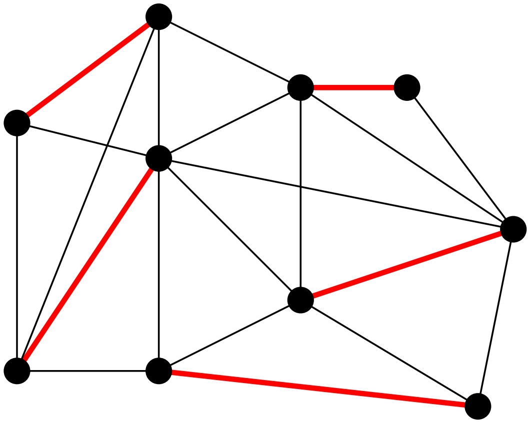
Minimum spanning tree implementations

We need to provide an edge vector to Kruskal's algorithm for storing MST edges.

Kruskal's algorithm

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

Recap: Maximum matching



- $G = (V, E)$
- $M \subseteq E$ 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

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

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

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

Setup: 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:
  1.61: This version is recommended (current Ubuntu LTS, Algolab VM).
  1.55+: These versions have Mincost-maxflow, should be fine.
- See the technical instructions page for more details.

Setup: 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++ -O3 -std=c++11 -I path/to/boost_1_61_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'

Setup: 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.

Setup: 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.

Setup: 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: 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: 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: 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 $\mathcal{O}(1)$.
- `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 in the source code

Useful stuff: Custom properties

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

```
enum edge_info_t { edge_info };
namespace boost {
    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] = ...
```

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:

```
// Prim non-named parameters example
prim_minimum_spanning_tree(G, startvertex,
make_iterator_property_map(predmap.begin(), get(vertex_index, G)),
make_iterator_property_map(distmap.begin(), get(vertex_index, G)),
get(edge_weight,G), get(vertex_index,G), default_dijkstra_visitor());
// Prim named parameters:
// PredecessorMap must be provided, all other parameters optional
prim_minimum_spanning_tree(G,
make_iterator_property_map(predmap.begin(), get(vertex_index, G)),
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.

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

- ...you use a pointer type as a property map (see e.g. here):
  Buggy for Dijkstra calls in combination with Strong components header.

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