A quick introduction to BGL

October 4, 2010

The contents of this document is based on the BGL website.

Creating a graph

For a more more detailed discussion of the following consult the following links: Using adjacency_list, adjacency_list reference, adjacency_matrix reference.

When creating a graph in BGL you have the choice between an adjacency matrix representation and an adjacency list representation. The latter is suited for most applications, the adjacency matrix should be used only for very dense graphs or if fast edge insertion and removal is crucial. Depending on your choice you need to instantiate one of the following two classes:

```
adjacency_list<OutEdgeList, VertexList, Directed, VertexProperties,
               EdgeProperties, GraphProperties, EdgeList>
adjacency_matrix<Directed, VertexProperties, EdgeProperties,
                 GraphProperty, Allocator>
```

The `OutEdgeList` and `VertexList` parameters are used to specify what data structure the class `adjacency_list` is using internally for storing outgoing edges and vertices respectively. The choices are `vecS`, `listS`, `slistS`, `setS`, `multisetS`, `hash_setS` for `std::vector`, `std::list`, `std::slist`, `std::set`, `std::multiset`, `std::hash_set`. Often `vecS` is a good choice for both. `EdgeList` is the same for the edge list of the graph.

The `Directed` parameter is usually either `directedS` or `undirectedS`. A third option is `bidirectionalS` (refer to the online manual for its description).

`VertexProperties`, `EdgeProperties` and `GraphProperties` each can be passed a property for all vertices, edges or the entire graph to have (default value is `no_property` for each of them). A property is defined as follows:

```
property<[property tag], [type], [optional: further property]>
```

for example

```
property<vertex_name_t, std::string>
```

if each vertex is to have a name of string value. Typical properties are weights, distances, capacities, etc. Multiple properties can be given by nesting them:

```
property<edge_weight_t, long, property<edge_capacity, long, ...> >
```

Different algorithms require different properties to be defined. Please consult the online documentation for the individual algorithms to find out which properties have to be defined. To improve the readability of your source code, you should define a type '`Graph`' hiding the above decisions. For shortest paths on a directed graph e.g., the definition might look something like

```
typedef adjacency_list<listS, vecS, directedS, no_property,
                       property<edge_weight_t, int> > Graph;
```

In the following, we will focus on the `adjacency_list` class as `adjacency_matrix` is rarely a good choice and works analogously.

Using adaptors for external data structures

Rather that creating a graph as described in the previous section, BGL also provides adaptors for alternate data structures. A useful example is defined in the header file `boost/graph/vector_as_graph.hpp`. This adaptor allows to use `std::vector<std::list<int> >` as graph, where the adjacency lists for the individual vertices are stored in a vector. The header file provides overloaded functions such that such a graph can seemlessly be used in the same ways as `adjacency_list` or `adjacency_matrix`.

Defining vertex and edge types

The first step when trying to access vertices and edges of a graph is to define the vertex and edge types. For this we need the 'traits' of the graph. Traits are general characteristics of the graph implementation like 'What is the type internally used for vertices/edges?'. There is two ways of accessing them for `adjacency_list`:

```
adjacency_list_traits<EdgeList, VertexList, Directed>
```

with parameters similar to the definition of the graph class, or

```
graph_traits<Graph>.
```

The former is usually used if `Graph` has not yet been defined, e.g. if we want to add the property `edge_reverse_t`, which requires the graph traits, to our `Graph` definition. Once we have defined `Graph`, we can use the second formulation. In both cases it is a best practice to introduce another `typedef` for a new type `Traits`.

Once the `Traits` class is specified, the vertex and edge types can be accessed via

```
Traits::vertex_descriptor
```

and

```
Traits::edge_descriptor.
```

Iterate over vertices and edges

To iterate over all vertices and edges of a graph `g`, we can do the following:

```
Traits::vertex_iterator vert_i, vert_end;
Traits::out_edge_iterator edge_i, edge_end;
for(tie(vert_i, vert_end) = vertices(g); vert_i != vert_end; ++vert_i)
  for(tie(edge_i, edge_end) = out_edges(*vert_i, g); edge_i != edge_end; ++edge_i)
    //do something (the vertices of the edge are *vert_i and target(*edge_i, g)
    //similarly the edge is *edge_i
```

In the above `vertices(g)` and `out_edges(*vert_i, g)` both return pairs of iterators. Rather than saving the pair and splitting it manually, BGL provides the function `tie()` which allows to assign two return values simultaneously.

Warning: the `add_edge(u,v,g)` operation invalidates all `out_edge_iterators` on `u` for directed graphs, and all `edge_iterators` on `u` **and** `v` in the case of undirected graphs. Be careful when using `add_edge()` in iterator loops, especially when the graph contains loop edges. See the Boost documentation for more information.

Property Maps

BGL relies on property maps for managing the properties of edges and vertices of the graph. A property map `pmap` is defined by

```
property_map<Graph, [property tag]>::type pmap;
```

Again the definition is usually hidden using `typedef`'s, e.g.

```
typedef property_map<Graph, edge_capacity_t>::type EdgeCapacityMap;
```

The property maps for properties the graph was defined to have for every vertex or edge are accessed via

```
get([property name], graph)
```

where the names of the properties are not to be confused with property tags. For the `EdgeCapacityMap` defined above, this would look like

```
EdgeCapacityMap cap_map = get(edge_capacity, graph);
```

The entries of a property map are accessed via

```
get([property map], [edge or vertex])
```

and modified via

```
put([property map], [edge or vertex], [value]).
```

The entries of a property map can also be accessed using `at()` or equivalently the `[]`-operator, e.g.

```
cap_map[edge] = 5
```

to assign a capacity of 5 to an edge.

All algorithms of BGL are passed the property maps for all properties they need to access explicitly.

Adding vertices and edges

There are two ways of creating vertices: we can either specify the size of the graph upon creation

```
Graph graph(size);
```

or add vertices one at a time

```
Traits::vertex_descriptor vert = add_vertex(graph);
```

Adding edges is similarly easy:

```
bool success;
Traits::edge_descriptor edge;
tie(edge, success) = add_edge(vert1, vert2, graph);
```

or

```
pair<Traits::edge_descriptor, bool> edge = add_edge(vert1, vert2, graph);
```

Here `vert1` and `vert2` may either be indices or vertex_descriptors.

Invoking BGL algorithms

Invoking algorithms is straight forward. Find the desired algorithm in the online documentation, check that your graph definition includes all required properties and call the corresponding function.

The graph characteristics ("graph concepts") that an algorithm requires are listed in the online documentation. Every algorithm lists the concepts a graph class has to model for the algorithm to work. Similarly the documentation of the graph class (`adjacency_list`, or an adaptor) lists the concepts it models. The requirements of the different concepts are explained separately.

In Detail I: Example for using Dijkstra's algorithm

In the following you find an example for using Dijkstra's algorithm.

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

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

using namespace std;
using namespace boost;

typedef adjacency_list<vecS, vecS, directedS, no_property,
                       property<edge_weight_t, int> > Graph;
typedef graph_traits<Graph>::vertex_descriptor Vertex;
typedef graph_traits<Graph>::edge_descriptor Edge;

const int NNODES = 5;
const int NEDGES = 8;
const int EDGES[NEDGES][3] = {{0, 1, 1}, {0, 2, 5}, {1, 2, 4}, {1, 3, 2},
                              {2, 4, 3}, {3, 2, 1}, {3, 4, 7}, {4, 0, 1}};
const int SOURCE = 0;
const int TARGET = 4;

int main()
{
  Graph g(NNODES);
  property_map<Graph, edge_weight_t>::type weightMap = get(edge_weight, g);
  for(int i = 0; i < NEDGES; ++i)
  {
    bool success;
    Edge e;
    tie(e, success) = add_edge(EDGES[i][0], EDGES[i][1], g);
    weightMap[e] = EDGES[i][2];
  }

  std::vector<Vertex> predecessors(num_vertices(g));
  std::vector<int> distances(num_vertices(g));

  //*/ //EITHER THIS ...
  dijkstra_shortest_paths(g, SOURCE,
    predecessor_map(&predecessors[0]).distance_map(&distances[0]));
```

```
  /*/ //... OR THIS ...
  dijkstra_shortest_paths(g, SOURCE, &predecessors[0], &distances[0],
                          get(edge_weight, g), get(vertex_index, g),
                          less<int>(), plus<int>(),
                           (numeric_limits<int>::max)(), 0,
                          default_dijkstra_visitor());
  //*/ //... END OR

  cout << "Calculated distances from " << SOURCE << ":" << endl;
  for(int i = 0; i < NNODES; ++i)
    cout << SOURCE << " to " << i << ": " << distances[i] << endl;
  cout << endl;

  cout << "Shortest path from " << SOURCE << " to " << TARGET << ":" << endl;
  cout << TARGET;
  int curr = TARGET;
  while(curr != SOURCE)
    cout << " <- " << (curr = predecessors[curr]);
  cout << endl;
}
```

In Detail II: Using flow algorithms

When applying a flow algorithm to a graph, BGL requires each edge to have a capacity, to have a residual capacity and to be mapped to its reverse edge. This means that usually the graph type is defined somewhat like the following:

```
typedef adjacency_list_traits<vecS, vecS, directedS> Traits;
typedef adjacency_list<vecS, vecS, directedS,
                       no_property,
                       property<edge_capacity_t, long,
                       property<edge_residual_capacity_t, long,
                       property<edge_reverse_t, Traits::edge_descriptor> > > > Graph;
```

When adding edges to the graph, these attributes need to be set. Below you find an outline of the typical procedure for adding an edge `e` from vertex `a` to vertex `b` with a capacity `c` from `a` to `b` in a graph `g`.

```
typedef property_map<Graph, edge_capacity_t>::type EdgeCapacityMap;
typedef property_map<Graph, edge_reverse_t>::type ReverseEdgeMap;
typedef property_map<Graph, edge_residual_capacity_t>::type ResidualCapacityMap;
typedef graph_traits<Graph>::edge_descriptor EdgeDescriptor;

EdgeCapacityMap capacity = get(edge_capacity, g);
ReverseEdgeMap rev_edge = get(edge_reverse, g);
ResidualCapacityMap res_capacity = get(edge_residual_capacity, g);

bool edgeDidNotExist;
EdgeDescriptor e, reverseE;
tie(e, edgeDidNotExist) = add_edge(a, b, g);
tie(reverseE, edgeDidNotExist) = add_edge(b, a, g);
capacity[e] = c;
capacity[reverseE] = 0;
```

```
rev_edge[e] = reverseE;
rev_edge[reverseE] = e;
```

Very often it is advisable to factor edge creation out into a function. This time we use `std::pair` explicitly, use whatever method you find more readable. In the following we have used two functions to enhance reusability. *Note that the capacity of the reverse edge must always be zero, if you intend to use the push-relable or Edmonds-Karp algorithms! (If you need a capacity in both directions, you need to create two edges with a zero-capacity reverse edge each)*

```
#include <cassert>
...
EdgeDescriptor createEdge(int from, int to, int c, Graph& g,
                          EdgeCapacityMap& capacity)
{
  pair<EdgeDescriptor, bool> edge = add_edge(from, to, g);
  assert(edge.second);
  capacity[edge.first] = c;
  return edge.first;
}

void createOneWayEdge(int from, int to, int c1, Graph& g,
                      EdgeCapacityMap& capacity, ReverseEdgeMap& rev_edge)
{
  EdgeDescriptor e = createEdge(from, to, c1, g, capacity);
  //REVERSE EDGE CAPACITY MUST BE 0
  EdgeDescriptor reverseE = createEdge(to, from, 0, g, capacity);
  rev_edge[e] = reverseE;
  rev_edge[reverseE] = e;
}
```

Once the graph is set up, you have the choice between the Edmonds-Karp and push-relabel algorithms. Push-relable is more efficient in all but the sparsest graphs. If you are only interested in the total flow, the following should be sufficient:

```
#include <boost/graph/push_relabel_max_flow.hpp>
long flow = push_relabel_max_flow(g, source, sink);
```

or

```
#include <boost/graph/edmonds_karp_max_flow.hpp>
long flow = edmonds_karp_max_flow(g, source, sink);
```

In Detail III: Finding a maximum cardinality matching

BGL unfortunately does not provide any algorithms for weighted matchings. For unweighted matchings, the graph configuration is straight forward:

```
#include <boost/graph/adjacency_list.hpp>
using namespace boost;

typedef adjacency_list<vecS, vecS, undirectedS> Graph;
typedef graph_traits<Graph>::vertex_descriptor VertexDescriptor;
```

Assume, we want to compute a maximum matching for the following graph.

```
const int NVERTICES = 8;
const int NEDGES = 8;
const int EDGES[NEDGES][2] = {{0, 7}, {1, 2}, {2, 3}, {2, 7},
                              {3, 4}, {3, 5}, {4, 5}, {6, 7}};
```

The graph setup, again, is straight forward:

```
Graph g(NVERTICES);
for(int i = 0; i < NEDGES; ++i)
  add_edge(EDGES[i][0], EDGES[i][1], g);
```

In order to invoke the maximum cardinality matching routine, we need a property map that will contain the resulting matching. We can simply use a `std::vector` to store for every vertex who its partner is, if any:

```
#include <boost/graph/max_cardinality_matching.hpp>
#include <vector>
using namespace std;
...
vector<VertexDescriptor> mate(NVERTICES);
edmonds_maximum_cardinality_matching(g, &mate[0]);
```

Depending on what you intend to do with the result, you can use `mate[i]` to access the vertex that was matched to vertex `i`. Vertices that were not matched are assigned a "null vertex". Assuming you want to output every pairing of the matching you could do it as follows:

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

Debugging BGL

One difficulty when using external libraries is that debugging can be tedious, because bugs are not always easy to pinpoint. Sadly, BGL is no exception to this rule. In this section we try to collect a few common issues when using BGL and how they can be addressed.

Problems with BGL often manifest through run-time crashes ("RUN-TIME ERROR" or "ASSERTION FAILED" in the judge). These crashes usually occur within a call to some BGL algorithm. However the problem is very often caused by the part of the source code that sets up the graph. You can narrow the suspicious source parts by gradually commenting out parts of it (not the algorithm call obviously!) and repeatedly submitting to the judge. This way you can often locate the problem.

Here is a list of common causes for problems:

1. *Problems with indexing during graph set-up.*
   If you insert edges with illegal indices into the graph, BGL will automatically try to resize the graph such that the indices become legal. If you add the edge $(0, 15)$ in a graph with 8 vertices, BGL will thus insert 7 new vertices automatically. To check whether you have made such an error, you can compare `num_vertices(g)` with the number of intended vertices after creating the graph.

2. *Problems with reverse edges etc.*
   If you are trying to use a flow algorithm, remember that the capacity of the reverse edges needs to be 0! Also make sure that you have reverse edges assigned to every edge.

3. *Problems when using iterators.*
   Be aware that iterators can become invalid if the graph is modified. See the warning in the section "Iterate over vertices and edges".

4. *Problems compiling your code.*
   Make sure that the graph class (or whatever) you chose models the required concepts for the algorithms you want to use. Edmonds-Karp maximum flow for instance requires your graph to be a model of `VertexListGraph` and `IncidenceGraph`. You can find the required concepts for each algorithm in the online documentation as well as a description of the idividual concepts themselves.

5. *Problems with the behavior of your code.*
   Find an example (as small as possible) where your code misbehaves. Output the graph to check whether you correctly built it. Simplify the graph as much as possible while trying to maintain the faulty behavior in order to locate the problem.