Algolab BGL Introduction

Akaki Mamageishvili

ETH Zürich

October 8, 2014

BGL

- Stands for *Boost Graph Library* (hence, part of Boost).
- (Extremely) generic C++ library of graph data structures and algorithms.
- Your new best friend – BGL docs (check the version number): `http://www.boost.org/doc/libs/1_56_0/libs/graph/doc/table_of_contents.html`

Installation

- Pre-installed in ETH computer rooms.
- System specific, sorry.
- A good thing though: there is no linking, headers only.
- On "standard" Linux distributions try getting a package from the repository.
- On MacOS package from MacPorts.

Prerequisites (and curriculum)

- Adjacency list vs. adjacency matrix.
- BFS & (also non-recursive) DFS.
- Topological sorting.
- Eulerian tours.
- Shortest paths.
- Minimum spanning trees.
- Strongly connected components.
- Biconnected components.
- Maximum flows and minimum cuts.
- Maximum matchings.
- Planarity testing.

Learn-as-you-go: Wikipedia, Cormen, Kleinberg etc.

Generic programming

```
void sort(int* beg, int* end);
void sort_descending(int* beg, int* end);
void sort(vector_int V);
void sort(double* beg, double* end);
...
```

A problem solved by *generic programming*. In particular, C++ features *templates*.

```
template<class Iterator, class Comparison>
void sort(Iterator beg, Iterator end, Comparison comp);
```

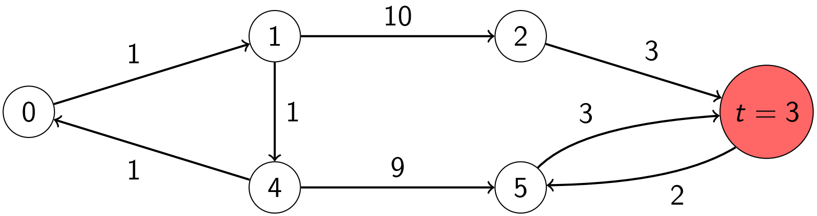
The idea of BGL is to do a similar thing for graph data types and algorithms.

Tutorial problem: statement

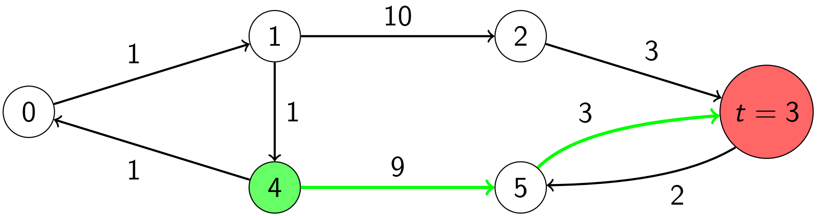
Input: A directed graph G with positive weights on edges and a vertex t.
Definition: We call a vertex u *universal* if all vertices in G can be reached from it.
Output: The length of a shortest path $u \to t$ that starts in some universal vertex u. If such a path does not exist, output `NO`.

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

Tutorial problem: example



Tutorial problem: example



Tutorial problem: how to start?

- "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! Model the problem, design the algorithm, understand why it should work, 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: theory

We say u and v are *strongly connected* if there exist paths $u \to v$ and $v \to u$.

Fact

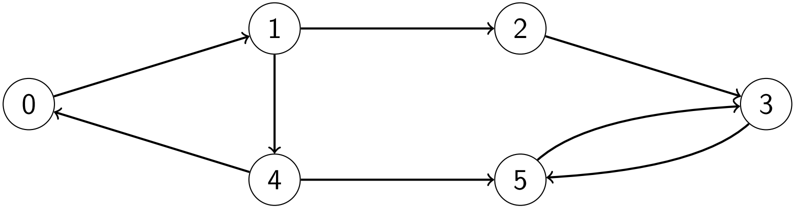
Being strongly connected is an equivalence relation.

We call an equivalence class of this relation a *strongly connected component*.

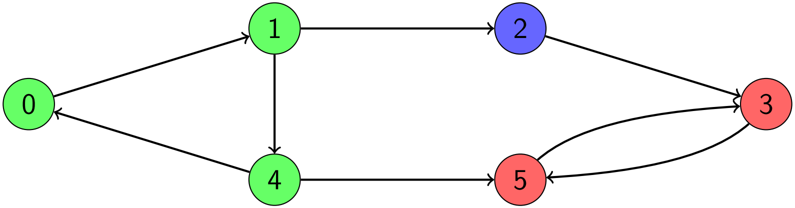
Fact

The (natural) graph of strongly connected components is a DAG.

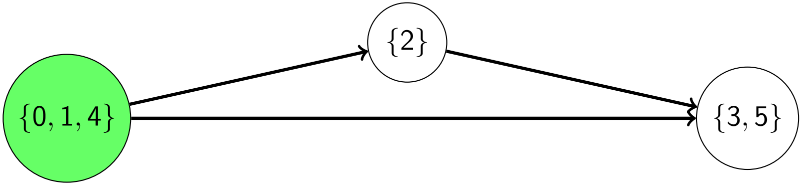
Tutorial problem: SCC example



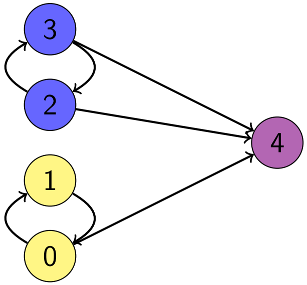
Tutorial problem: SCC example

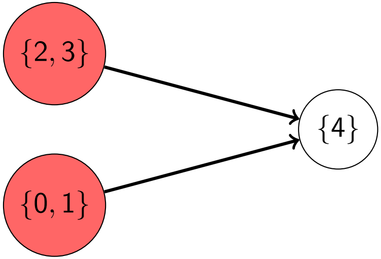


Tutorial problem: SCC DAG example


$$\begin{tikzcd}
 & \{2\} \arrow[dr] & \\
\{0,1,4\} \arrow[ur] \arrow[rr] & & \{3,5\}
\end{tikzcd}$$

Tutorial problem: SCC example





Tutorial problem: modeling

Let us call a strongly connected component with no in-edges in SCC DAG a *minimal component*.

Fact

If there is more than one minimal component in G, then there it does not have a universal u.

Lemma

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

Tutorial problem: modeling

New formulation of the problem:

1. If there exists > 1 minimal strongly connected component in G, output `NO`.
2. Output the shortest distance $u \rightarrow t$ for 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 store a graph? Hint: 95% of the time adjacency_list is a good choice.
- strong_components.
- How to check how many maximal components are there? topological_sort? Maybe there is a simple ad hoc?
- 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

```
// Directed graph with int weights on edges.
typedef adjacency_list<vecS, vecS, directedS,
   no_property,
   property<edge_weight_t, int> > Graph;
// Edge type (edge descriptor in BGL speak).
typedef graph_traits<Graph>::edge_descriptor Edge;
// Edge iterator.
typedef graph_traits<Graph>::edge_iterator
    EdgeIterator;
// 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: no. of testcases.
    cin >> T;
    while (T--) testcase();
}
```

Tutorial problem: code – input

```
void testcase() {
   int V, E, s; // 1st line: <ver_no> <edg_no> <src>
   cin >> V >> E >> s;
   Graph G(V);
   WeightMap weight_map = 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;
      tie(e, tuples::ignore) =
         add_edge(v, u, G);  // Create G_T directly.
      weight_map[e] = c;
    }  // for
```

Tutorial problem: code – strong components

```
void testcase() {
    ...
    vector<int> scc(V);
    int nscc = strong_components(G, &scc[0]);
```

Tutorial problem: code – maximal SCCs

```
void testcase() {
    ...
    // vector<int> instead of vector<bool>.
    vector<int> is_max(nscc, true);
    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);
        // Why does it work? Exercise.
        if (scc[u] != scc[v]) is_max[scc[u]] = false;
    } // for
```

Tutorial problem: code – maximal SCCs

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

Tutorial problem: code – Dijkstra

```
void testcase() {
    ...
    vector<int> dist(V);
    dijkstra_shortest_paths(
        G, s, distance_map(&dist[0]));
```

Tutorial problem: output

```cpp
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 << "\n";
} // testcase()
```

Disclaimer

This is one way of doing things. There are others. There are also numerous ways of hurting yourself with BGL.

Namespace

Simple: everything is in namespace `boost`.

```
using namespace boost;
```

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);
```

Graph types: adjacency list

This is the class you almost always need.

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

- `vecS` — for each vertex, adjacency list kept in a `vector`. Other choices: `listS`, `setS`.
- `vecS` — list of edges kept in a `vector`.
- `directedS` — directed graph. Other choice: `undirectedS`.

Graph types: adjacency matrix

Makes sense for dense graphs. Simpler syntax.

```
#include <boost/graph/adjacency_matrix.hpp>
...
typedef adjacency_matrix<directedS> Graph;
```

Graph types

- **Self-loops**: In principle allowed. Some algorithms might misbehave. Better to avoid them.
- **Parallel edges**: Most algorithms can handle them. Allowed for `adjacency_list` unless `setS` is used for the first template parameter.

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);
```

Creating a graph

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

- Constructs an empty graph with `n` vertices.

Adding an edge

```
Graph G(n);
...
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.
- Amortized $O(1)$ in most cases.

Removing vertices and edges

```
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.

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.

Iterating over all vertices

```
for (int u = 0; u < num_vertices(G); ++u) {
   // do something
}
```

- Simple.

Iterating over all 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.

Iterating over edges incident to a vertex

```
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.

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.

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.

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 weight_map = get(edge_weight, G);
weight_map[e] = k;
```

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

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.

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 {
   ...
};
```

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 info_map = get(edge_info, G);
info_map[e] = ...
```

Calling an algorithm

- Example: `kruskal_minimum_spanning_tree`.
- Follow the doc page.
- Header:
  `#include <boost/graph/kruskal_min_spanning_tree.hpp>`
- BGL uses the concept of *concepts*. A class is a *model* of a given concept if it implements certain operations and has certain properties. For Kruskal MST, a graph type must be a model of Vertex List Graph and Edge List Graph. `adjacency_list` is a model of both of them (in fact, it works for almost all algorithms).

Calling an algorithm

```
vector<Edge> E(num_edges(G));
vector<Edge>::iterator ebeg, eend =
   kruskal_minimum_spanning_tree(G, E.begin());
for (ebeg = E.begin(); ebeg != eend; ++ebeg) {
   ...
}
```

- Second argument is an iterator where the edges will be output, so it could also be e.g., a pointer to a (big enough) array.
- Return value is the end of the range of the output edges.
- `edge_weight_t` map must be defined for this to work.

Calling an algorithm – named parameters

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

- Sometimes you need to provide additional custom arguments. This is done via *named parameters*.
- To pass a `vector` as a vertex property map you need to provide `&V[0]`, an iterator like `V.begin()` will not work.
- `rank_map` and `predecessor_map` give you additional information computed by union-find algorithm.
- Defaults: `get(edge_weight, G)` for `weight_map`, internally created `vectors` for `rank_map` and `predecessor_map`.

Problems – compilation

Error messages can be terrible.

- Consider re-compiling the code after every line written. This will help to identify the problem quickly.
- There will be confusing `typedefs`, nested types, iterators etc. Come up with a naming pattern and stick to it.

Problems – runtime

- 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 mistake in your edge insertion.

Problem of the week

As usual, on Monday. Be advised it doesn't have to be BGL. Anything already covered in the course can be used.

The end

GOOD LUCK!

Other documents

- BGL docs: `http://www.boost.org/doc/libs/1_56_0/libs/graph/doc/table_of_contents.html`.
- Another quick introduction on Moodle.