BGL Introduction

Charlotte Knierim

Some of the material by:
Andreas Bärtschi, Petar Ivanov, Chih-Hung Liu, Martin Raszyk, and Daniel Wolleb

What is BGL?

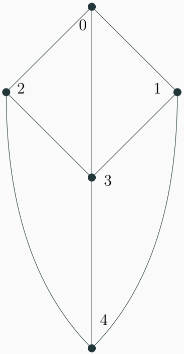
- Library of graph algorithms
- Documentation is available on https://algolab.inf.ethz.ch/doc/.
- Solve problems using graphs without having to implement standard algorithms

Roadmap

- **BGL Introduction**
  - Declaring and initializing a graph in BGL
  - Examples of standard graph algorithms in BGL
  - Tutorial problem: from a problem statement to a full solution
- Flows
- Advanced flows

Graph definition

We represent a graph $G = (V, E)$ as an adjacency list



Space $O(n + m)$

Vertex	List of neighbors
0	[1, 2, 3]
1	[0, 3, 4]
2	[0, 3, 4]
3	[0, 1, 2, 4]
4	[1, 2, 3]

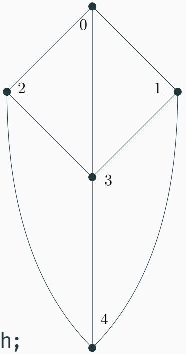
STL vs BGL

C++ Standard Library

```
#include <vector>

typedef std::vector<int>          neighbor_list;
typedef std::vector<neighbor_list>  cpp_graph;
```

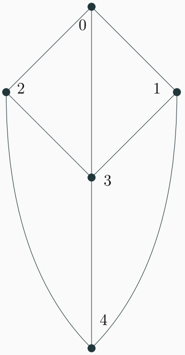
BGL

```
#include <boost/graph/adjacency_list.hpp>

typedef boost::adjacency_list<boost::vecS,
                              boost::vecS,
                              boost::undirectedS> graph;
```



Initializing the graph

```
void init_graph(){
    graph G(5);

    boost::add_edge(0, 1, G);
    boost::add_edge(0, 2, G);
    boost::add_edge(0, 3, G);
    boost::add_edge(1, 3, G);
    boost::add_edge(1, 4, G);
    boost::add_edge(2, 3, G);
    boost::add_edge(2, 4, G);
    boost::add_edge(3, 4, G);

    }
```



Warning!

`boost::add_edge(0, 7, G);` would extend the vertex set of `G` to eight vertices!

Iterate over the Edges

all edges:

```
typedef boost::graph_traits<graph>::edge_iterator edge_it;

edge_it e_beg, e_end;
for (boost::tie(e_beg, e_end) = boost::edges(G); e_beg != e_end; ++e_beg) {
    std::cout << boost::source(*e_beg, G) << " "
                                   << boost::target(*e_beg, G) << "\n";}
```

Warning: Be careful with iterators when removing edges!

neighbors of a vertex:

```
typedef boost::graph_traits<graph>::out_edge_iterator out_edge_it;

out_edge_it oe_beg, oe_end;
for (boost::tie(oe_beg, oe_end) = boost::out_edges(0, G);
                                             oe_beg != oe_end; ++oe_beg) {
    assert(boost::source(*oe_beg, G) == 0);
    std::cout << boost::target(*oe_beg, G) << "\n";}
```

Other graphs types

Undirected graphs

```
typedef boost::adjacency_list<boost::vecS,
                              boost::vecS,
                              boost::directedS> directed_graph;
```

Weighted graphs

```
typedef boost::adjacency_list<boost::vecS,
                              boost::vecS,
                              boost::directedS,
                              boost::no_property,
                              // no vertex property
                              boost::property<boost::edge_weight_t, int>
                              // interior edge weight property
                              > weighted_graph;
```

Predefined Vertex and Edge Properties

Some predefined vertex and edge properties:

- `vertex_degree_t`
- `vertex_name_t`
- `vertex_distance_t`
- `edge_weight_t`
- `edge_capacity_t`
- `edge_residual_capacity_t`
- `edge_reverse_t`

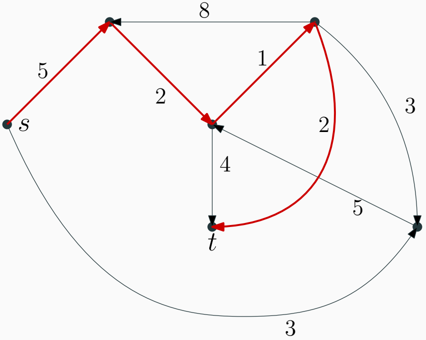
All property maps must be initialized and maintained **manually**!

Examples of standard graph algorithms in BGL

1. Shortest path in weighted, directed graphs using Dijkstra’s Algorithm
2. Minimum spanning tree in weighted, undirected graphs using Kruskal’s Algorithm
3. Maximum matching in unweighted, undirected graph using Edmond’s Algorithm

Problem: shortest path between two vertices

Input: a directed, weighted graph $G = (V, E)$, vertices $s, t \in V$
Output: distance between s and t



Distance between Two Vertices: Dijkstra's Algorithm

```
#include <boost/graph/dijkstra_shortest_paths.hpp>

int dijkstra_dist(const weighted_graph &G, int s, int t) {
    int n = boost::num_vertices(G);
    std::vector<int> dist_map(n);

    boost::dijkstra_shortest_paths(G, s,
        boost::distance_map(boost::make_iterator_property_map(dist_map.begin(),
                                                  boost::get(boost::vertex_index, G))));

    return dist_map[t];
}
```

Time complexity of `boost:dijkstra_shortest_paths` is $O(n \log n + m)$

Reconstructing the path

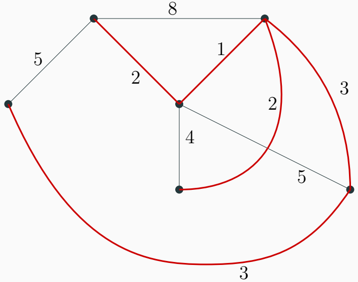
What if we also want to keep track of the path?

```
typedef boost::graph_traits<weighted_graph>::vertex_descriptor vertex_desc;

int dijkstra_path(const weighted_graph &G, int s, int t,
                                    std::vector<vertex_desc> &path) {
    int n = boost::num_vertices(G);
    std::vector<int>         dist_map(n); std::vector<vertex_desc> pred_map(n);

    boost::dijkstra_shortest_paths(G, s,
        boost::distance_map(boost::make_iterator_property_map(dist_map.begin(
        boost::get(boost::vertex_index, G)))
        .predecessor_map(boost::make_iterator_property_map(pred_map.begin(),
        boost::get(boost::vertex_index, G))));

    int cur = t;
    path.clear(); path.push_back(cur);
    while (s != cur) {
        cur = pred_map[cur]; path.push_back(cur);}
    std::reverse(path.begin(), path.end());
    return dist_map[t];}}
```

Problem: Minimum Spanning Tree

A minimum spanning tree of a connected, undirected, weighted graph $G = (V, E)$ is a spanning subtree of minimum weight (sum of the weights of the edges)

Input: a connected, undirected, weighted graph $G = (V, E)$
Output: an edge set $E' \subseteq E$ that forms the MST



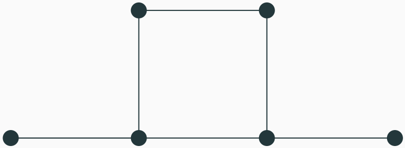
Minimum Spanning Tree: Kruskal's Algorithm

```
#include <boost/graph/kruskal_min_spanning_tree.hpp>

typedef boost::adjacency_list<boost::vecS, boost::vecS, boost::undirectedS,
                              boost::no_property,
                              boost::property<boost::edge_weight_t, int>
                              > weighted_graph;

typedef boost::graph_traits<weighted_graph>::edge_descriptor      edge_desc;

void kruskal(const weighted_graph &G) {
    std::vector<edge_desc> mst;    // vector to store MST edges (not a property ma

    boost::kruskal_minimum_spanning_tree(G, std::back_inserter(mst));

    for (std::vector<edge_desc>::iterator it = mst.begin(); it != mst.end(); ++it
        std::cout << boost::source(*it, G) << " " << boost::target(*it, G) << "\n
```

Time complexity of `boost:kruskal_minimum_spanning_tree` is $O(m \log m)$.

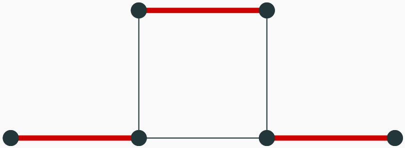
Problem: Maximum matching

Input: an undirected (unweighted!) graph $G = (V, E)$
Output: a set $M \subseteq E$ such that M is a matching in G and $|M|$ is maximal



Problem: Maximum matching

Input: an undirected (unweighted!) graph $G = (V, E)$
Output: a set $M \subseteq E$ such that M is a matching in G and $|M|$ is maximal

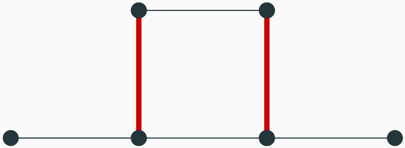


Maximum (perfect) matching in G

Problem: Maximum matching

Input: an undirected (unweighted!) graph $G = (V, E)$

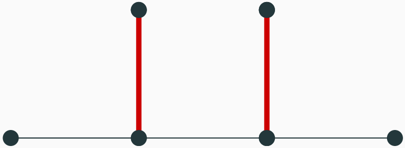
Output: a set $M \subseteq E$ such that M is a matching in G and $|M|$ is maximal



Maximal (but not maximum) matching in G

Problem: Maximum matching

Input: an undirected (unweighted!) graph $G = (V, E)$
Output: a set $M \subseteq E$ such that M is a matching in G and $|M|$ is maximal



not every graph has a perfect matching

Maximum Matching: Edmond's Algorithm

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

void maximum_matching(const graph &G) {
    int n = boost::num_vertices(G);
    std::vector<vertex_desc> mate_map(n);  // exterior property map
    const vertex_desc NULL_VERTEX = boost::graph_traits<graph>::null_vertex();

    boost::edmonds_maximum_cardinality_matching(G,
            boost::make_iterator_property_map(mate_map.begin(),
            boost::get(boost::vertex_index, G)));
    int matching_size = boost::matching_size(G,
            boost::make_iterator_property_map(mate_map.begin(),
            boost::get(boost::vertex_index, G)));

    for (int i = 0; i < n; ++i) {
        if (mate_map[i] != NULL_VERTEX && i < mate_map[i])
            std::cout << i << " " << mate_map[i] << "\n";}}
```

Time complexity of `boost::edmonds_maximum_cardinality_matching` is $O(mn \cdot \alpha(m,n))$.

Tutorial problem: Formal Problem Statement

Input: A directed, unweighted graph $G = (V, E)$

Output: All universal vertices in G

First approach

How do we test if a given vertex $v \in V$ is universal?

$\Longrightarrow$ start a BFS in v, if it visits all vertices $\rightarrow v$ is universal

The run time of this is $O(n+m)$.

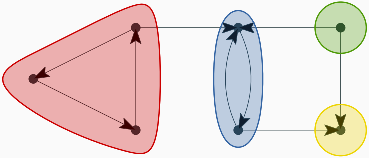
Find all universal vertices in $O(n(n+m))$.

Code

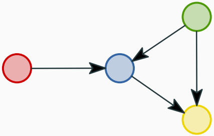
```
#include <boost/graph/breadth_first_search.hpp>
#include <boost/graph/properties.hpp>

typedef boost::adjacency_list<boost::vecS, boost::vecS, boost::directedS> graph;
typedef boost::default_color_type                                         color;
const color black = boost::color_traits<color>::black(); // visited by BFS
const color white = boost::color_traits<color>::white(); // not visited by BFS

bool is_universal(const graph &G, int u) { // Is u universal in G?
    int n = boost::num_vertices(G);
    std::vector<color> vertex_color(n); // exterior property map

    boost::breadth_first_search(G, u,
        boost::color_map(boost::make_iterator_property_map(
            vertex_color.begin(), boost::get(boost::vertex_index, G))));

    // u is universal iff no vertex is white
    return (std::find(vertex_color.begin(), vertex_color.end(), white)
                                                    == vertex_color.end());
}
```

Strong connected components

A strongly connected component (SCC) of a directed graph $G = (V, E)$ is any maximal subset of vertices $C \subseteq V$ such that all vertices in C are pairwise reachable (via directed paths).



A directed graph G



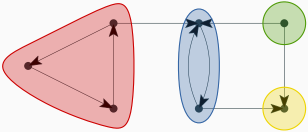
The condensation of G (acyclic)

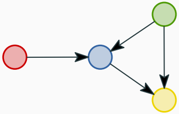
We call a SCC a source SCC if it has no incoming edges from other SCC

Second approach – How do SCC help with the problem?

If there is exactly one source SCC, all vertices in this SCC are universal (and no other vertices are universal) Why?

- Every non-empty directed graph has a source SCC
- A vertex in a SCC can reach all the vertices in the same component
- If there is a unique source SCC its corresponding vertex is universal in the condensation of G
- This implies that all vertices in the unique source SCC are universal in G
- No vertex from outside the source SCC can reach a vertex inside the source SCC (and can thus not be universal)





Second approach – Algorithm

1. Calculate the SCCs of G
2. Check, which SCCs are source SCCs
3. If there is more than one source SCC $\implies$ there is no universal vertex
4. Else there is exactly one source SCC and we output all vertices belonging to this SCC

Tutorial Problem: Full Solution - Build the graph

```
#include <boost/graph/adjacency_list.hpp>
#include <boost/graph/strong_components.hpp>

typedef boost::adjacency_list<boost::vecS, boost::vecS, boost::directedS> graph;
typedef boost::graph_traits<graph>::edge_iterator                         edge_it;

void testcase() {
    int n, m;
    std::cin >> n >> m;
    graph G(n);

    for (int i = 0; i < m; ++i) {
        int u, v;
        std::cin >> u >> v;
        boost::add_edge(u, v, G);
    }
```

Tutorial Problem: Full Solution — Strongly Connected Components

```cpp
// scc_map[i]: index of SCC containing i-th vertex
std::vector<int> scc_map(n);  // exterior property map
// nscc: total number of SCCs
int nscc = boost::strong_components(G,
    boost::make_iterator_property_map(scc_map.begin(),
    boost::get(boost::vertex_index, G)));
```

Time complexity of `boost::strong_components` is $O(n+m)$.

Tutorial Problem: Full Solution – Source SCCs

```
// is_src[i]: is i-th SCC a source?
std::vector<bool> is_src(nscc, true);
edge_it ebeg, eend;

for (boost::tie(ebeg, eend) = boost::edges(G); ebeg != eend; ++ebeg) {
    int u = boost::source(*ebeg, G), v = boost::target(*ebeg, G);
    // edge (u, v) in G implies that component scc_map[v] is not a source
    if (scc_map[u] != scc_map[v]) is_src[scc_map[v]] = false;
}
```

Time complexity $O(m)$

Tutorial Problem: Full Solution – Finding All Universal Vertices

```
    int src_count = std::count(is_src.begin(), is_src.end(), true);
    if (src_count > 1) { // no universal vertex among multiple SCCs
        std::cout << "\n";
    return;
    }
    assert(src_count == 1);
    // recall property of the condensation DAG (directed acyclic graph)

    // all vertices in the single source SCC are universal
    for (int v = 0; v < n; ++v) {
        if (is_src[scc_map[v]]) std::cout << v << " ";
    }
    std::cout << "\n";
} /* end of function testcase */
```

Time complexity of `testcase` is $O(n + m)$.

Overview

The following algorithms can appear in exercises. Please familiarize yourself with them. This list is non exhaustive and will be extended throughout the course.

Algorithm	Runtime
`boost::breadth_first_search`	$O(n+m)$
`boost::depth_first_search`	$O(n+m)$
`boost::dijkstra_shortest_path`	$O(n \log n + m)$
`boost::kruskal_minimum_spanning_tree`	$O(m \log m)$
`boost::connected_components`	$O(n+m)$
`boost::strong_components`	$O(n+m)$
`boost::biconnected_components`	$O(n+m)$
`boost::articulation_points`	$O(n+m)$
`boost::edmonds_maximum_cardinality_matching`	$O(mn \cdot \alpha(m,n))$
`boost::is_bipartite`	$O(n+m)$

What next?

- Familiarize yourself with BGL
- Read up on theory if something today was new to you
- We provide some very easy problems to get used to the typedefs
  – also code snippets