BGL Introduction

Anouk Paradis

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

What is BGL?

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

Roadmap

- **BGL Introduction**
- Flows
- Advanced flows

Declaring Graphs in BGL

Examples of standard graph algorithms in BGL

Tutorial Problem

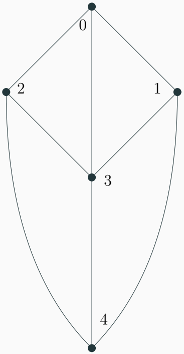
Overview

Graph definition

We represent a graph $G = (V, E)$ as an adjacency list. G has n vertices and m edges.

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]



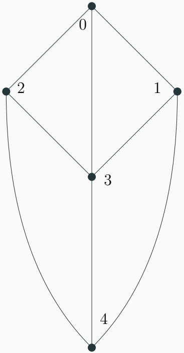
STL vs BGL

C++ Standard Library

```
#include <vector>

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

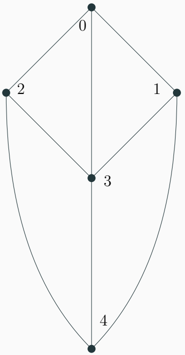
BGL

```
#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";}
```

For G undirected, `out_edges` is all incident edges.

Other graphs types

Directed 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>  // edge property (interior)
                              > 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**!

Declaring Graphs in BGL

Examples of standard graph algorithms in BGL

Tutorial Problem

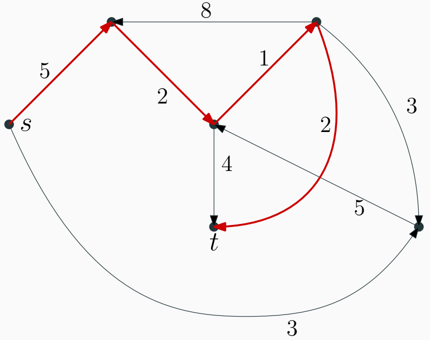
Overview

Examples of standard graph algorithms in BGL

1. Shortest path using Dijkstra's Algorithm
2. Minimum spanning tree using Kruskal's Algorithm
3. Maximum matching using Edmond's Algorithm
4. Strongly connected components using Tarjan'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



Recall: Dijkstra's algorithm is one to all

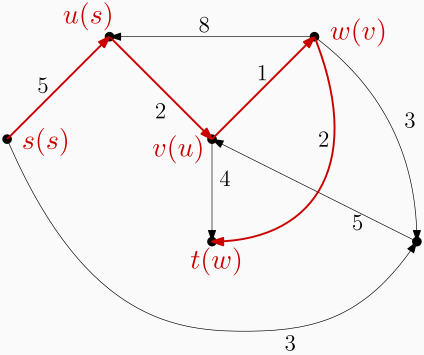
Distance between Two Vertices: Dijkstra's Algorithm

```
#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); //exterior property

    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?
$\rightarrow$remember for each vertex the "previous step"



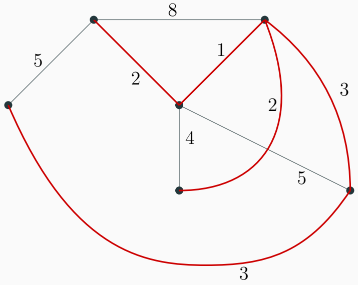
Reconstructing the path

```
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

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

an acyclic subgraph of G connecting all vertices in V and having the minimum sum of edge weights



Works with negative weights.

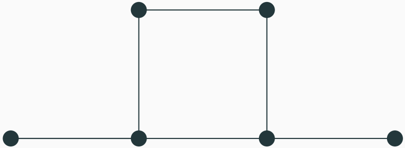
Minimum Spanning Tree: Kruskal's Algorithm

```
#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 map!)

    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 $O(m \log m)$. Uses Union Find data structure - also available in boost

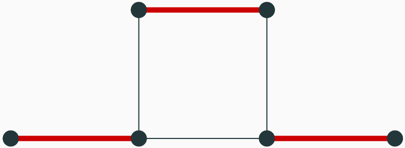
Problem: Maximum matching

Input: an undirected unweighted graph $G = (V, E)$
Output: a set of edges $M \subseteq E$ such that $|M|$ is maximum and no two edges in M share any endpoint.



Problem: Maximum matching

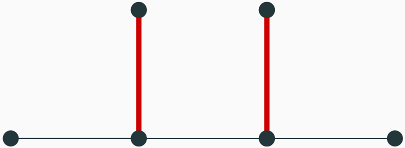
Input: an undirected unweighted graph $G = (V, E)$
Output: a set of edges $M \subseteq E$ such that $|M|$ is maximum and no two edges in M share any endpoint.



Maximum (perfect) matching in G

Problem: Maximum matching

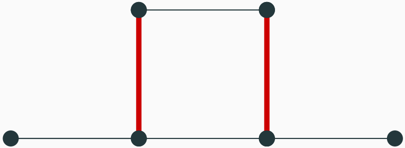
Input: an undirected unweighted graph $G = (V, E)$
Output: a set of edges $M \subseteq E$ such that $|M|$ is maximum and no two edges in M share any endpoint.



not every graph has a perfect matching

Problem: Maximum matching

Input: an undirected unweighted graph $G = (V, E)$
Output: a set of edges $M \subseteq E$ such that $|M|$ is maximum and no two edges in M share any endpoint.



Warning! Greedy may fail: a maximal matching is not always maximum

Maximum Matching: Edmond's Algorithm

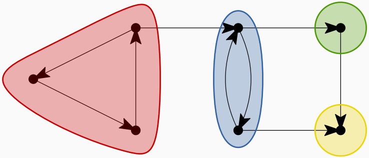
```
#include <boost/graph/max_cardinality_matching.hpp>

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 $O(mn \cdot \alpha(m,n))$ (Remember: $\alpha(m,n) \leq 4$).

Problem: Strongly Connected Components

A strongly connected component 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.

Input: a directed, unweighted graph $G = (V, E)$
Output: the number of strongly connected components in G



Strongly Connected Components: Tarjan’s Algorithm

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

void strong_connected_comp(const graph &G) {
    int n = boost::num_vertices(G);

    std::vector<int> scc_map(n);  // exterior property map

    int nscc = boost::strong_components(G,
        boost::make_iterator_property_map(scc_map.begin(),
        boost::get(boost::vertex_index, G)));

    std::cout << "Number of connected components: " << nscc << "\n";
    for (int i = 0; i < n; ++i) {
        std::cout << i << " " << scc[i] << "\n";}
```

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

Tutorial problem: Universal Warehouses

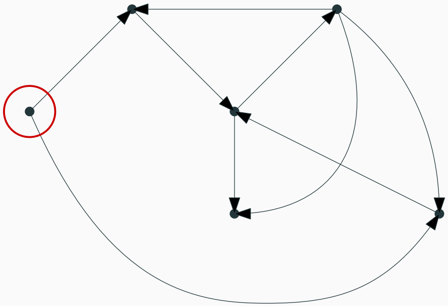
B-city is made of multiple locations, linked by unidirectional roads. Alice wants to create a delivery empire, able to deliver anywhere in the city. For this she needs to decide where to build her warehouse. She wants it to be universal: any point in the city must be reachable from this warehouse. To make the best decision, she asks you to find all possible warehouse locations, that is to say all universal locations in the city.

Constraints

1s
$0 \leq n \leq 5.10^4$ number of locations
$0 \leq m \leq 5.10^4$ number of roads.

Tutorial problem: Universal Warehouses

B-city is made of multiple locations, linked by unidirectional roads. Alice wants to create a delivery empire, able to deliver anywhere in the city. For this she needs to decide where to build her warehouse. She wants it to be universal: any point in the city must be reachable from this warehouse. To make the best decision, she asks you to find all possible warehouse locations, that is to say all universal locations in the city.



This vertex is the only universal vertex in this graph.

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?

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

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());
}
```

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 $\to v$ is universal

Complexity?

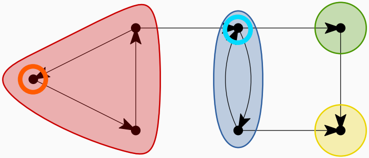
For each vertex: $O(n+m)$.
Altogether: $O(n(n+m))$.

For $n \leq 5.10^4$ and $m \leq 5.10^4$
we get $\sim 5.10^{10} \gg 10^7$.
$\Longrightarrow$ too slow

Second approach

How could we "group" vertices instead of checking them individually?

Recall strongly connected components:



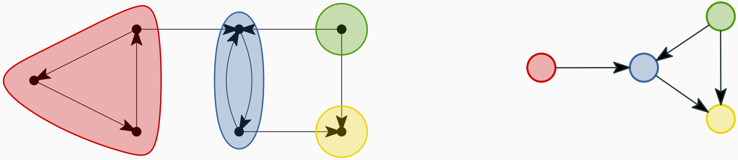
If u is universal, so is its strongly connected component.

If u can reach v, then u can reach any node in v strongly connected component.

$\Longrightarrow$ we can work directly on the strongly connected components

Second approach

Working on the strongly connected components: condensation of G



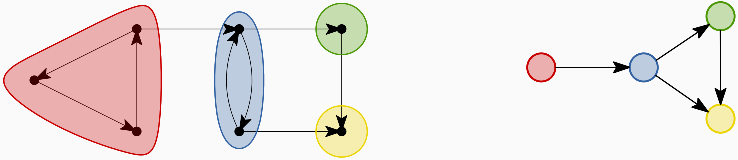
The condensation of G is acyclic.

$\implies$ there are source SCC.

If more than one source SCC: no universal nodes.

Second approach

Working on the strongly connected components: condensation of G



The condensation of G is acyclic.

$\implies$ there are source SCC.

If more than one source SCC: no universal nodes.

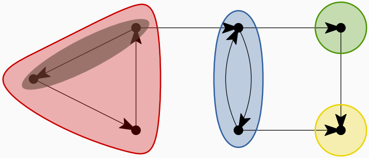
Else (exactly 1 source SCC): all its vertices are universal.

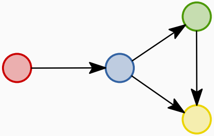
Second approach – Algorithm

1. Compute the SCCs of G
2. Check which SCCs are source SCCs
3. If there is more than one source SCC $\Longrightarrow$ no universal vertex
4. Else there is exactly one source SCC $\Longrightarrow$ all vertices in this SCC

Second approach – Algorithm

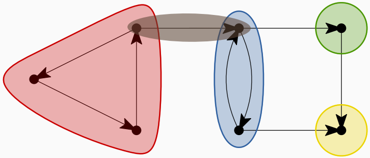
1. Compute the SCCs of G
2. Check which SCCs are source SCCs

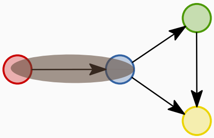




Second approach – Algorithm

1. Compute the SCCs of G
2. Check which SCCs are source SCCs





Second approach – Algorithm

1. Compute the SCCs of $G \implies O(n+m)$
2. Check which SCCs are source SCCs $\implies O(m)$
3. If there is more than one source SCC $\implies$ no universal vertex
4. Else there is exactly one source SCC $\implies$ all vertices in this SCC

Complexity?

Altogether: $O(n+m)$.

For $n \leq 5.10^4$ and $m \leq 5.10^4$
we get $\sim 10^5 < 10^7$.
$\implies$ it fits!

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 – Source SCCs

```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)));

// 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;
}
```

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 */
```

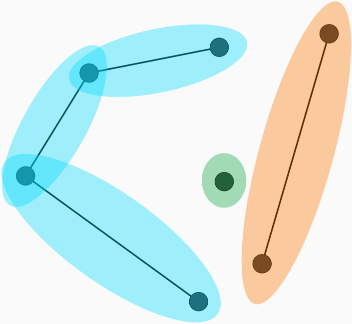
Overview

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::edmonds_maximum_cardinality_matching`	$O(mn\cdot\alpha(m,n))$
`boost::strong_components`	$O(n+m)$
`boost::connected_components`	$O(n+m)$
`boost::biconnected_components`	$O(n+m)$
`boost::articulation_points`	$O(n+m)$
`boost::is_bipartite`	$O(n+m)$

Connected Components

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



Biconnected Components

A biconnected graph is an undirected graph that is connected, and remains connected even if a vertex is removed. A biconnected component is any maximal subgraph of G that is biconnected.

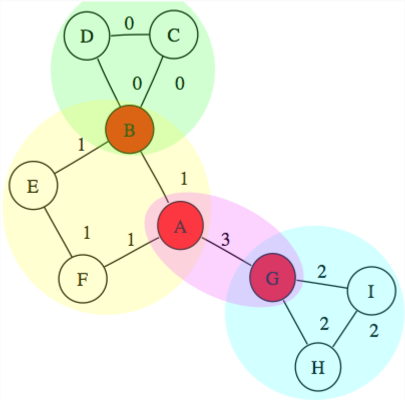


Image from boost documentation

Articulation Points

An articulation point of a undirected graph is any vertex part of two biconnected components.

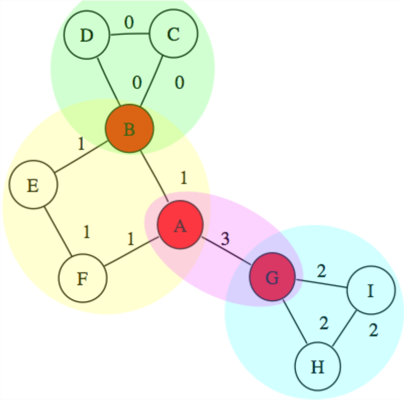
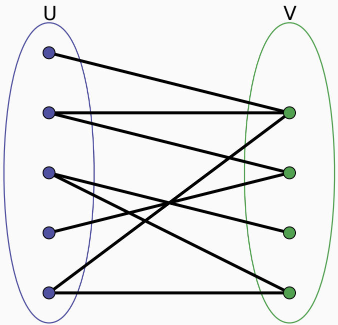


Image from boost documentation

Bipartite Graph

A graph $G = (V, E)$ is bipartite if V can be split in two subsets U, V such that all edges in E have an extremity in each.



What next?

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