Algolab Graph and BGL Introduction

Martin Raszyk[1]

October 9, 2019

[1] using material from Andreas Bärtschi, Petar Ivanov, Chih-Hung Liu, and Daniel Wolleb

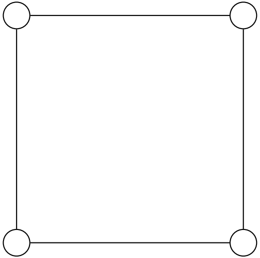
Roadmap

- definition of a graph and its representations
- declaring and initializing a graph in BGL
- examples of standard graph algorithms in BGL
- tutorial problem: from a problem statement to a full solution

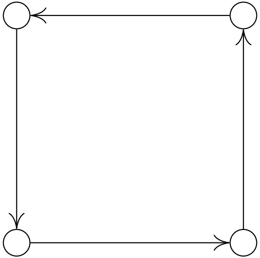
Graph: Definition

Definition

A *graph* $G = (V, E)$ consists of a set of n *vertices* (nodes) V and m *edges* E.



Undirected graph.



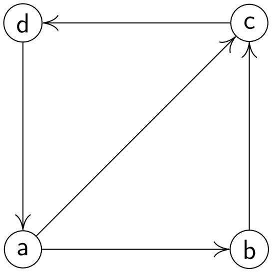
Directed graph.

Directed Graph: Representation

Adjacency matrix:

$$\begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

Space complexity: $\Theta(n^2)$.



Adjacency list: USE THIS REPRESENTATION!

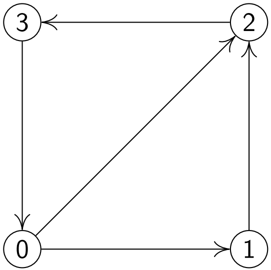
Vertex	List of neighbors
a	[b, c]
b	[c]
c	[d]
d	[a]

Space complexity: $\Theta(n + m)$.

Adjacency List in C++ Standard Library

```cpp
#include <vector>

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

vertex	neighbor_list
0	[1, 2]
1	[2]
2	[3]
3	[0]



Adjacency List in BGL

- C++ Standard Library

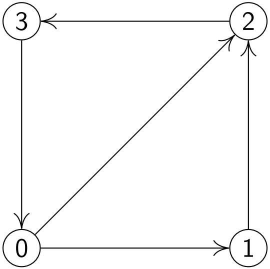
```
#include <vector>

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::directedS> graph;
```

Warm-up: Initialize a Graph. . .

```
void init_graph()
{
  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::add_edge(0, 2, G);
```



WARNING!

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

Warm-up: . . . and Iterate over its 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";
}
```

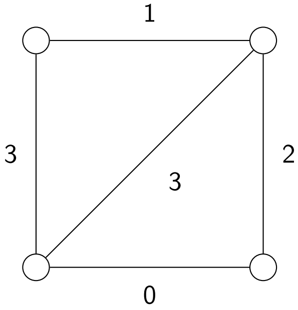
- neighbors of a vertex:

```
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";
}
} /* end of function init_graph */
```

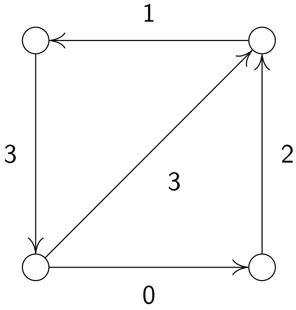
Weighted Graph: Definition

Definition

A *weighted* graph $G = (V, E, w)$ consists of a set of vertices V, edges E, and a *weight function* $w : E \to \mathbb{Z}$.



Undirected weighted graph.



Directed weighted graph.

Weighted Graph in BGL

Definition

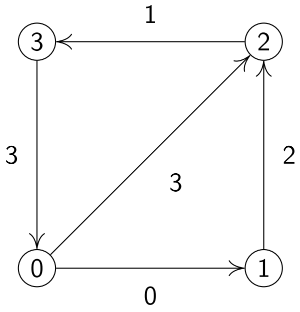
A *weighted* graph $G = (V, E, w)$ consists of a set of vertices V, edges E, and a *weight function* $w : E \to \mathbb{Z}$.

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

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;

typedef boost::property_map<weighted_graph, boost::edge_weight_t>::type weight_map;
```

Initialize a Weighted Graph

```
typedef boost::property_map<weighted_graph, boost::edge_weight_t>::type weight_map;
typedef boost::graph_traits<weighted_graph>::edge_descriptor edge_desc;

void init_weighted_graph()
{
  weighted_graph G(4);
  weight_map weights = boost::get(boost::edge_weight, G);

  edge_desc e;
  e = boost::add_edge(0, 1, G).first; weights[e]=0;
  e = boost::add_edge(1, 2, G).first; weights[e]=2;
  e = boost::add_edge(2, 3, G).first; weights[e]=1;
  e = boost::add_edge(3, 0, G).first; weights[e]=3;
  e = boost::add_edge(0, 2, G).first; weights[e]=3;
}
```

- edge_weight_t: a C++ type
- edge_weight: a C++ value



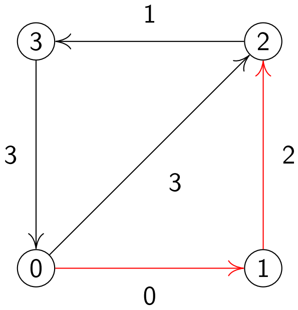
Predefined Vertex and Edge Properties

Some *predefined* vertex and edge properties:

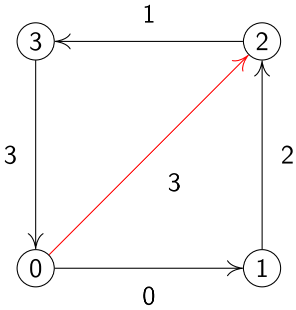
- `vertex_degree_t`
- `vertex_distance_t`
- `edge_weight_t`
- `edge_capacity_t`
- `edge_residual_capacity_t`
- `edge_reverse_t`

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

Shortest Path between Two Vertices



Shortest path from 0 to 2.



Longer path from 0 to 2.

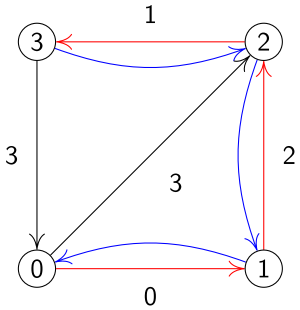
Distance between Two Vertices: Dijkstra's Algorithm

```
#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 $O(n \log n + m)$.

Shortest Path between Two Vertices: Dijkstra's Algorithm

```
#include <boost/graph/dijkstra_shortest_paths.hpp>
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)))
/* dot! */ .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];
}
```

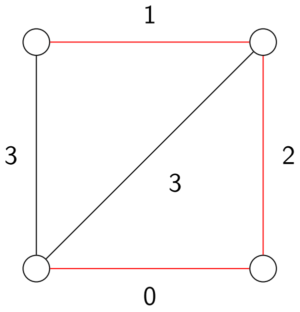


Shortest path edges are red.
Predecessor edges are blue.

Minimum Spanning Tree

Definition

A *minimum spanning tree* of a connected undirected weighted graph $G = (V, E)$ is an acyclic subgraph of G connecting all vertices in V and having the minimum sum of edge 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";
  }
}
```

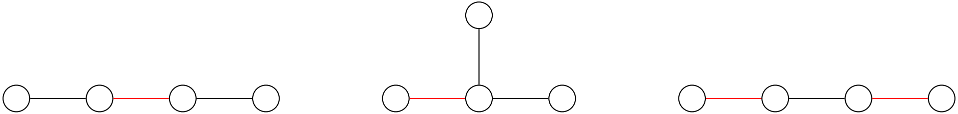
Time complexity of `boost::kruskal_minimum_spanning_tree` is $O(m \log m)$.

Maximum Matching

Definition

A *maximum matching* in an undirected graph $G = (V, E)$ is a subset of its edges with maximum cardinality such that no two edges in the matching share any endpoint.

Remark. A *maximal matching* (i.e., one which cannot be further extended) can be obtained by a simple greedy algorithm. It is *not* necessarily a maximum matching!



Maximal matching. Maximum matching. Maximum (perfect) matching.

Maximum Matching: Edmond's Algorithm

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

typedef boost::adjacency_list<boost::vecS, boost::vecS, boost::undirectedS> graph;
typedef boost::graph_traits<graph>::vertex_descriptor                        vertex_desc;

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) {
    // mate_map[i] != NULL_VERTEX: the vertex is matched
    // i < mate_map[i]: visit each edge in the matching only once
    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 $O(mn \cdot \alpha(m, n))$.

BGL Outlook – More Algorithms Available!

To be covered in the remainder of this tutorial:

- breadth-first search (`breadth_first_search`)
- strongly connected components in directed graphs (`strong_components`)

To be covered in upcoming tutorials:

- maximum-flow algorithms
- minimum-cost maximum-flow algorithms

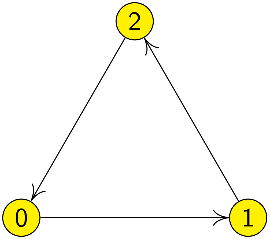
Exercise:

- connected components in undirected graphs (`connected_components`)
- check if a graph is bipartite (`is_bipartite`)
- minimum spanning tree (`kruskal_minimum_spanning_tree`, `prim_minimum_spanning_tree`)

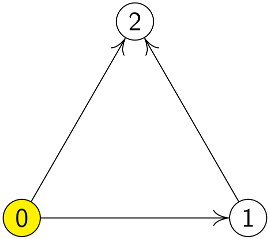
Tutorial Problem: Universal Vertex

Definition

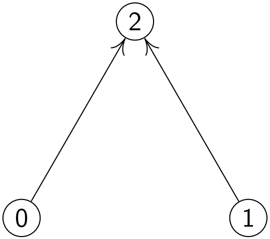
Let $G = (V, E)$ be a directed graph. We call a vertex $u \in V$ *universal* if every other vertex $v \in V$ can be reached from u via a directed path.



Any vertex is universal.



Only 0 is universal.

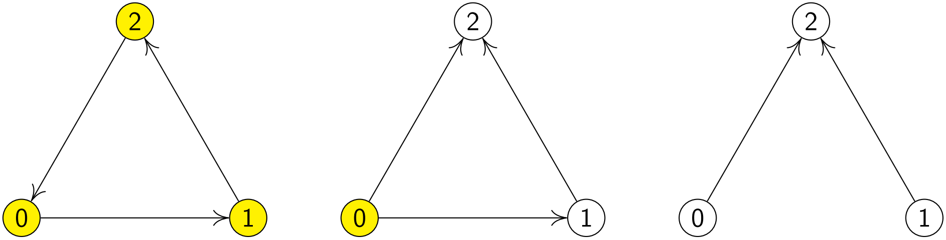


No vertex is universal.

Tutorial Problem: The Problem Statement

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

Output All universal vertices in G.



$\{0, 1, 2\}$ $\{0\}$ $\{\}$

Tutorial Problem: Checking if a Vertex is Universal

```
#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 (i.e., all vertices are reachable from u)
  return (std::find(vertex_color.begin(), vertex_color.end(), white) == vertex_color.end());
}
```

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

Tutorial Problem: A Slow Solution

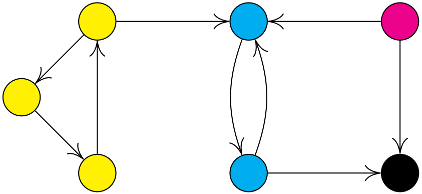
```
void testcase() {
  int n, m;
  std::cin >> n >> m;
  graph G(n);

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

  for (int i = 0; i < n; ++i) {
    if (is_universal(G, i)) std::cout << i << " ";
  }
  std::cout << "\n";
}
```

Time complexity of `testcase` is $O(n^2 + nm)$.

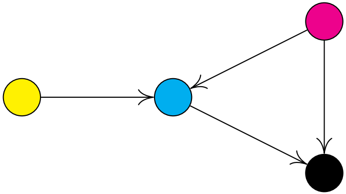
Tutorial Problem: Strongly Connected Components

Definition

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.

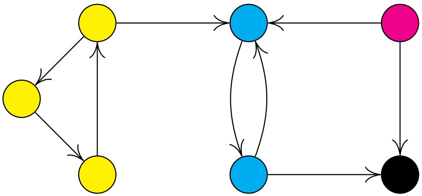
Lemma

The condensation of a directed graph is acyclic.

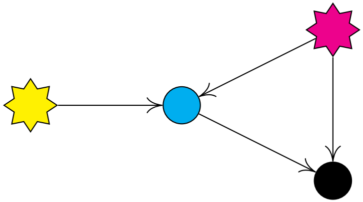
Tutorial Problem: Source SCC

Definition

A strongly connected component is called a *source* if it has no incoming edges from other SCCs.



A directed graph G.



The *condensation* of G.

Lemma

Any non-empty directed graph has a source SCC.

Lemma

If a directed graph has a single source SCC, then it is a universal vertex in its condensation.

Tutorial Problem: Full Solution (I)

```
#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 (II) — Strongly Connected Components

```
// 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 (III) – 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;
}
```

Tutorial Problem: Full Solution (IV) – 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 $O(n + m)$.

Tutorial Problem: Full Solution (V) – Main Function

```cpp
int main()
{
    int T;
    std::cin >> T;

    while(T--) testcase();

    return 0;
}
```