

# AlgoLab Graph and BGL Introduction

Chih-Hung Liu, slides from Daniel Wolleb, Petar Ivanov, Andreas Bärtschi

October 10, 2018

# Graph definitions

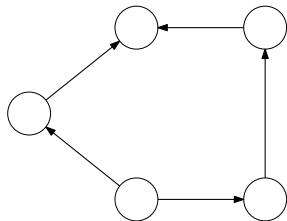
A graph has  $n$  vertices/nodes  $V$  and  $m$  edges/arcs  $E$ .

- ▶ (un-) directed
- ▶ (non-) weighted
- ▶ (a-) cyclic
- ▶ (dis-) connected

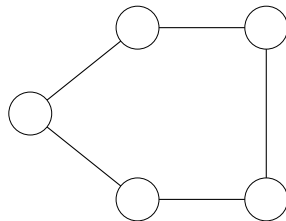
# Graph definitions

A graph has  $n$  vertices/nodes  $V$  and  $m$  edges/arcs  $E$ .

- ▶ (un-) directed
- ▶ (non-) weighted
- ▶ (a-) cyclic
- ▶ (dis-) connected



Directed

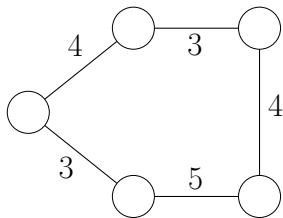


Undirected

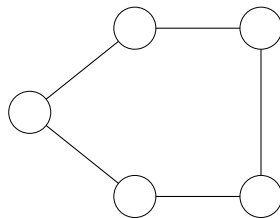
# Graph definitions

A graph has  $n$  vertices/nodes  $V$  and  $m$  edges/arcs  $E$ .

- ▶ (un-) directed
- ▶ (non-) weighted
- ▶ (a-) cyclic
- ▶ (dis-) connected



Weighted

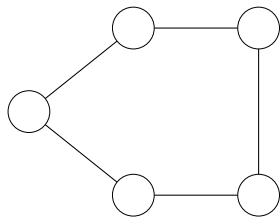


Non-weighted

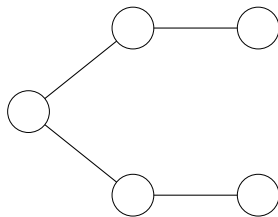
# Graph definitions

A graph has  $n$  vertices/nodes  $V$  and  $m$  edges/arcs  $E$ .

- ▶ (un-) directed
- ▶ (non-) weighted
- ▶ (a-) cyclic
- ▶ (dis-) connected



Cyclic

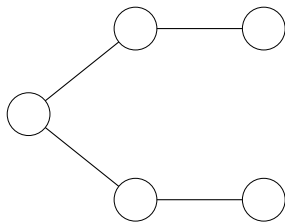


Acyclic

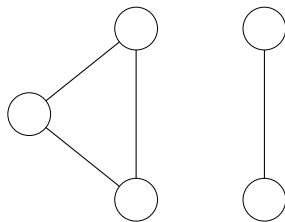
# Graph definitions

A graph has  $n$  vertices/nodes  $V$  and  $m$  edges/arcs  $E$ .

- ▶ (un-) directed
- ▶ (non-) weighted
- ▶ (a-) cyclic
- ▶ (dis-) connected



Connected



Disconnected

# Complexity-driven programming (not yet a real thing):

- ▶  $\Theta(V + E)$  – great!  $E < 10^{7\dots9}$
- ▶  $\Theta(V \cdot \log(V + E))$  – cool
- ▶  $\Theta(V \cdot E)$  – maybe ok
- ▶  $\Theta(2^V)$  – slow,  $V < 20 \dots 40$

General note:

- ! approach<sup>Find shortest paths</sup>  $\neq$  algorithm<sup>Dijkstra</sup>  $\neq$  implementation<sup>with adj.matrix</sup>
- ! abstract data type<sup>Dictionary</sup>  $\neq$  data structure<sup>Red-Black tree</sup>  $\neq$  implementation<sup>as in STL</sup>



Boost  
Graph  
Library

A **generic** C++ library of graph data structures and algorithms.

**BGL docs** – your new best friend:

[https://www.boost.org/doc/libs/1\\_68\\_0/libs/graph/doc/index.html](https://www.boost.org/doc/libs/1_68_0/libs/graph/doc/index.html)

and also on the judge:

[https://judge.inf.ethz.ch/doc/algolab/bgl/boost\\_1\\_58\\_0/libs/graph](https://judge.inf.ethz.ch/doc/algolab/bgl/boost_1_58_0/libs/graph)

Moodle: There's a brief **copy & paste manual**.

Algolab VM & General: There's a [technical instructions page](#) for all things Algolab.



# BGL: A generic library

Genericity type	STL	BGL
Algorithm / Data-Structure Interoperability	Decoupling of algorithms and data-structures Key ingredients: iterators	Decoupling of graph algorithms and graph representations Vertex iterators, edge iterators, adjacency iterators
Parameterization	Element type parameterization	Vertex and edge property multi-parameterization Associate <i>multiple</i> properties Accessible via <i>property maps</i>
Extensions (not covered in Algotab)	through function objects	through a <i>visitor object</i> , event points and methods depend on particular algorithm

# BGL: A generic library

Genericity type	STL	BGL
Algorithm / Data-Structure In- teroperability	Decoupling of algorithms and data-structures Key ingredients: iterators	Decoupling of graph algorithms and graph representations Vertex iterators, edge iterators, adjacency iterators
Parameterization	Element type parameterization	Vertex and edge property multi-parametrization Associate <i>multiple</i> properties Accessible via <i>property maps</i>
Extensions (not covered in Algotab)	through function objects	through a <i>visitor object</i> , event points and methods depend on particular algorithm

# BGL: Graph Representations

Representation	Advantages	Do
Adjacency list	Swiss army knife: Directed/undirected graphs, allow/disallow parallel-edges, efficient insertion, fast adjacency structure exploitation	<b>use this!</b>
Adjacency matrix	Dense graphs	<i>use at your own risk!</i>

# BGL: adjacency\_list

Example **without** any vertex or edge properties:

```
1 // Easy syntax. Parameters:
2 // OutEdgeList type, VertexList type, Directivity
3 typedef boost::adjacency_list<boost::vecS, boost::vecS, boost::directedS>    Graph;
4
5 // which is the same as:
6 typedef boost::adjacency_list<boost::vecS, boost::vecS, boost::directedS,
7 boost::no_property,          // the graph has no interior vertex properties
8 boost::no_property          // the graph has no interior edge properties
9 >                               Graph;
```

## BGL: adjacency\_list

Example **without** any vertex or edge properties:

```
1 // Easy syntax. Parameters:
2 // OutEdgeList type, VertexList type, Directivity
3 typedef boost::adjacency_list<boost::vecS, boost::vecS, boost::directedS>    Graph;
4
5 // which is the same as:
6 typedef boost::adjacency_list<boost::vecS, boost::vecS, boost::directedS,
7 boost::no_property,          // the graph has no interior vertex properties
8 boost::no_property          // the graph has no interior edge properties
9 >                               Graph;
```

Defines a *directed* Graph where the vertices are stored in a vector (VertexList vecS) and the outgoing edges in each vertex are stored in a vector (OutEdgeList vecS).

(Also see *Useful stuff: Options for adjacency\_list*, page 44.)

# BGL: adjacency\_list

Example **with** vertex property and multiple edge properties:

```
1 // Note the nested syntax for defining more than one edge property
2 typedef boost::adjacency_list<boost::vecS, boost::vecS, boost::directedS,
3 boost::property<boost::vertex_name_t, string>,           // interior vertex property
4 boost::property<boost::edge_capacity_t, int>,             // interior edge properties
5 boost::property<boost::edge_residual_capacity_t, int>, // nested syntax
6 boost::property<boost::edge_reverse_t, Traits::edge_descriptor> > > > Graph;
7
8 typedef boost::property_map<Graph, boost::vertex_name_t>::type NameMap;
9 typedef boost::property_map<Graph, boost::edge_capacity_t>::type CapacityMap;
10 typedef boost::property_map<Graph, boost::edge_residual_capacity_t>::type ResidualMap;
11 typedef boost::property_map<Graph, boost::edge_reverse_t>::type ReverseMap;
```

## BGL: adjacency\_list

Example **with** vertex property and multiple edge properties:

```
1 // Note the nested syntax for defining more than one edge property
2 typedef boost::adjacency_list<boost::vecS, boost::vecS, boost::directedS,
3 boost::property<boost::vertex_name_t, string>,           // interior vertex property
4 boost::property<boost::edge_capacity_t, int>,             // interior edge properties
5 boost::property<boost::edge_residual_capacity_t, int>,    // nested syntax
6 boost::property<boost::edge_reverse_t, Traits::edge_descriptor> > > > Graph;
7
8 typedef boost::property_map<Graph, boost::vertex_name_t>::type      NameMap;
9 typedef boost::property_map<Graph, boost::edge_capacity_t>::type    CapacityMap;
10 typedef boost::property_map<Graph, boost::edge_residual_capacity_t>::type ResidualMap;
11 typedef boost::property_map<Graph, boost::edge_reverse_t>::type     ReverseMap;
```

Interior properties are stored with the graph. Property Maps allow us to access the interior properties of the graph. Think of these as a mapping (object with operator `[]`). Also see *Useful stuff: Interior property maps, pages 48–49*.

## Warm-up: Read a Graph

```
7 typedef boost::adjacency_list<vecS, vecS, directedS> Graph;
8
9 int main()
10 {
11     Graph G(4);
12
13     boost::add_edge(0, 1, G);
14     boost::add_edge(1, 2, G);
15     boost::add_edge(2, 3, G);
16     boost::add_edge(3, 0, G);
17
18     boost::graph_traits<Graph>::edge_iterator ebegin, eend;
19     for(boost::tie(ebegin, eend) = boost::edges(G); ebegin != eend; ++ebegin){
20         std::cout << "G contains an edge from " << boost::source(*ebegin, G)
21                 << "to " << boost::target(*ebegin, G) << std::endl;
22     }
23 }
```



## Warm-up: Read a weighted Graph

```
7 typedef boost::adjacency_list<vecS, vecS, directedS, no_property,  
8                               property<edge_weight_t, int> > Graph;  
9 typedef boost::property_map<Graph, edge_weight_t>::type WeightMap;  
10 typedef Graph::edge_descriptor Edge;  
11  
12  
13 int main()  
14 {  
15     Graph G(4);  
16     WeightMap weights = boost::get(edge_weight, G);  
17     bool added;  
18     Edge e;  
19     boost::tie(e, added) = boost::add_edge(0, 1, G); weights[e]=1;  
20     boost::tie(e, added) = boost::add_edge(1, 2, G); weights[e]=3;  
21     boost::tie(e, added) = boost::add_edge(2, 3, G); weights[e]=-2;  
22     boost::tie(e, added) = boost::add_edge(3, 0, G); weights[e]=5;  
23  
24     graph_traits<Graph>::edge_iterator ebegin, eend;  
25     for(boost::tie(ebegin, eend) = boost::edges(G); ebegin != eend; ++ebegin)  
26         std::cout << "Edge (" << boost::source(*ebegin, G) << ", " << boost::target(*ebegin, G)  
27             << ") has weight " << weights[*ebegin] << std::endl;  
28 }
```

# BGL: Graph Algorithms

Area	Topic	Details
Basics	Distances	Dijkstra shortest paths
	Components	Kruskal minimum spanning tree Connected, biconnected & strongly connected components
	General Matchings	General unweighted matching
Flows	Maximum Flow	Graph setup (residual graph) Edmonds-Karp and Push-Relabel
	Disjoint paths	Vertex- / Edge-disjoint s-t paths
Advanced Flows	Minimum Cut	Maxflow-Mincut Theorem
	Bipartite Matchings	Vertex Cover & Independent Set
	Mincost Maxflow	Bipartite weighted matching & more

# BGL: Graph Algorithms

Area	Topic	Details
Basics	Distances	Dijkstra shortest paths
	Components	Kruskal minimum spanning tree
	General Matchings	Connected, biconnected & strongly connected components General unweighted matching
Flows	Maximum Flow	Graph setup (residual graph) Edmonds-Karp and Push-Relabel
	Disjoint paths	Vertex- / Edge-disjoint s-t paths
Advanced Flows	Minimum Cut	Maxflow-Mincut Theorem
	Bipartite Matchings	Vertex Cover & Independent Set
	Mincost Maxflow	Bipartite weighted matching & more

Many more (not in Algotab 2018): planarity testing, sparse matrix ordering, ...

# BGL: Graph Algorithms

Area	Topic	Details
Basics	Distances	Dijkstra shortest paths
	Components	Kruskal minimum spanning tree Connected, biconnected & strongly connected components
	General Matchings	General unweighted matching
Flows	Maximum Flow	Graph setup (residual graph) Edmonds-Karp and Push-Relabel
	Disjoint paths	Vertex- / Edge-disjoint s-t paths
Advanced Flows	Minimum Cut	Maxflow-Mincut Theorem
	Bipartite Matchings	Vertex Cover & Independent Set
	Mincost Maxflow	Bipartite weighted matching & more

Many more (not in Algotab 2018): planarity testing, sparse matrix ordering, ...

**Prerequisites:** Graph theory, BFS, DFS, topological sorting, Eulerian tours, Union-Find...

# Recap: Graph Traversal and Shortest paths

Graph Traversal with vertices partitioned into three sets: visited, enqueued, unknown

- ▶ **BFS** – closest first,  $\mathcal{O}(V + E)$
- ▶ **DFS** – furthest first,  $\mathcal{O}(V + E)$
- ▶ **Dijkstra** – weighted closest first,  $\mathcal{O}(E + V \log V)$  ([BGL docs](#))

Shortest paths with negative weights: Induction on number of edges in subpaths

- ▶ **Bellman Ford** – shortest paths from a single source,  $\mathcal{O}(V \cdot E)$  ([BGL docs](#))
- ▶ **Floyd–Warshall** – all-pairs shortest paths,  $\mathcal{O}(V^3)$  ([BGL docs](#))

Both can also be used to detect negative cycles

# Recap: Components (Vertex Connectivity)

Undirected Graph:

- ▶ **Connected** – vertex pair with a path
- ▶ **Biconnected** – vertex pair in a cycle
- ▶ **Articulation points** – vertex disconnecting a component
- ▶ **Bridges** – edge disconnecting a component

Directed Graph:

- ▶ **Strongly connected** – vertex pair with directed path in both ways

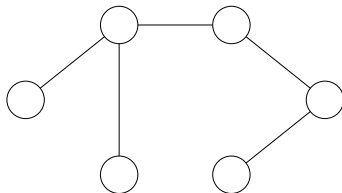
## Recap: Components (Vertex Connectivity)

Undirected Graph:

- ▶ **Connected** – vertex pair with a path
- ▶ **Biconnected** – vertex pair in a cycle
- ▶ **Articulation points** – vertex disconnecting a component
- ▶ **Bridges** – edge disconnecting a component

Directed Graph:

- ▶ **Strongly connected** – vertex pair with directed path in both ways



Connected

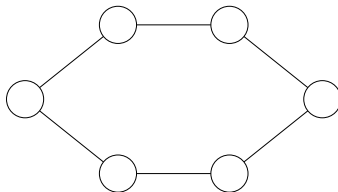
## Recap: Components (Vertex Connectivity)

Undirected Graph:

- ▶ **Connected** – vertex pair with a path
- ▶ **Biconnected** – vertex pair in a cycle
- ▶ **Articulation points** – vertex disconnecting a component
- ▶ **Bridges** – edge disconnecting a component

Directed Graph:

- ▶ **Strongly connected** – vertex pair with directed path in both ways



Bi-connected



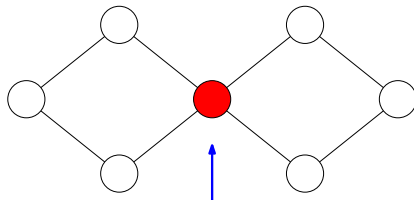
# Recap: Components (Vertex Connectivity)

Undirected Graph:

- ▶ **Connected** – vertex pair with a path
- ▶ **Biconnected** – vertex pair in a cycle
- ▶ **Articulation points** – vertex disconnecting a component
- ▶ **Bridges** – edge disconnecting a component

Directed Graph:

- ▶ **Strongly connected** – vertex pair with directed path in both ways



Articulation Point

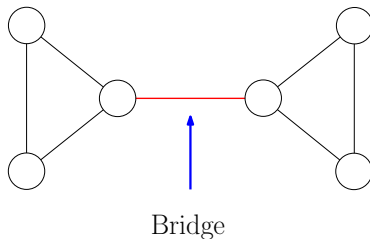
# Recap: Components (Vertex Connectivity)

Undirected Graph:

- ▶ **Connected** – vertex pair with a path
- ▶ **Biconnected** – vertex pair in a cycle
- ▶ **Articulation points** – vertex disconnecting a component
- ▶ **Bridges** – edge disconnecting a component

Directed Graph:

- ▶ **Strongly connected** – vertex pair with directed path in both ways



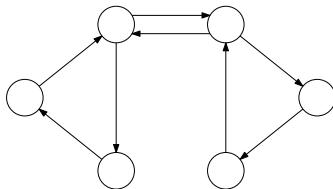
# Recap: Components (Vertex Connectivity)

Undirected Graph:

- ▶ **Connected** – vertex pair with a path
- ▶ **Biconnected** – vertex pair in a cycle
- ▶ **Articulation points** – vertex disconnecting a component
- ▶ **Bridges** – edge disconnecting a component

Directed Graph:

- ▶ **Strongly connected** – vertex pair with directed path in both ways



Strongly Connected

## Tutorial problem: statement & example

**Input** A directed graph  $G$  with positive weights on edges and a vertex  $t$ ,  
 $|V(G)| \leq 10^5$ ,  $|E(G)| \leq 2 \cdot 10^5$ .

**Definition** We call a vertex  $u$  *universal* if all vertices in  $G$  can be reached from it.

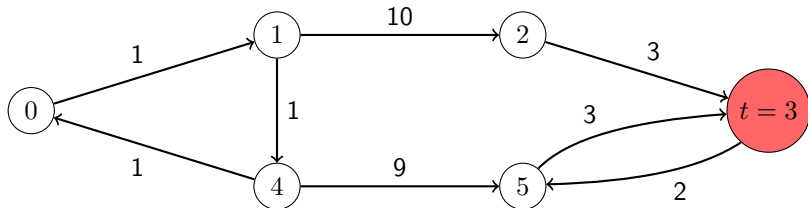
**Output** Minimum length of a shortest path  $u \rightarrow t$  over all universal vertices  $u$ .  
If such a path does not exist, output NO.

## Tutorial problem: statement & example

**Input** A directed graph  $G$  with positive weights on edges and a vertex  $t$ ,  
 $|V(G)| \leq 10^5$ ,  $|E(G)| \leq 2 \cdot 10^5$ .

**Definition** We call a vertex  $u$  *universal* if all vertices in  $G$  can be reached from it.

**Output** Minimum length of a shortest path  $u \rightarrow t$  over all universal vertices  $u$ .  
If such a path does not exist, output NO.

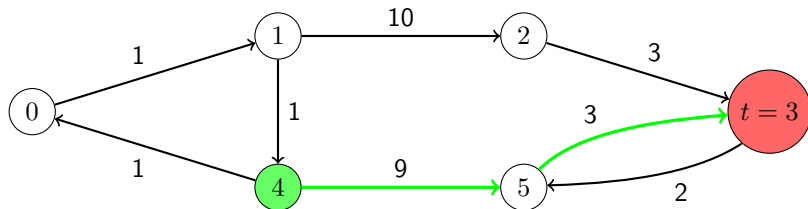


## Tutorial problem: statement & example

**Input** A directed graph  $G$  with positive weights on edges and a vertex  $t$ ,  
 $|V(G)| \leq 10^5$ ,  $|E(G)| \leq 2 \cdot 10^5$ .

**Definition** We call a vertex  $u$  *universal* if all vertices in  $G$  can be reached from it.

**Output** Minimum length of a shortest path  $u \rightarrow t$  over all universal vertices  $u$ .  
If such a path does not exist, output NO.



# Tutorial problem: how to start?

Time's short, so hurry up!

# Tutorial problem: how to start?

Time's short, so hurry up!

- ▶ "Check if there is a unique  $u$  with no in-edges, if yes output shortest path  $u \rightarrow t$ ." (**what if there is no such  $u$ ?**)



# Tutorial problem: how to start?

Time's short, so hurry up!

- ▶ "Check if there is a unique  $u$  with no in-edges, if yes output shortest path  $u \rightarrow t$ ." (**what if there is no such  $u$ ?**)
- ▶ "For each  $u$  check with DFS if  $u$  reaches all vertices, then..." (**too slow**)

# Tutorial problem: how to start?

Time's short, so hurry up!

- ▶ "Check if there is a unique  $u$  with no in-edges, if yes output shortest path  $u \rightarrow t$ ." (**what if there is no such  $u$ ?**)
- ▶ "For each  $u$  check with DFS if  $u$  reaches all vertices, then..." (**too slow**)
- ▶ Start coding:

```
1 #include <iostream>
2 int main() {
3     // some random algorithm
4 }
```

# Tutorial problem: how to start?

Time's short, so hurry up!

- ▶ "Check if there is a unique  $u$  with no in-edges, if yes output shortest path  $u \rightarrow t$ ." (**what if there is no such  $u$ ?**)
- ▶ "For each  $u$  check with DFS if  $u$  reaches all vertices, then..." (**too slow**)
- ▶ Start coding:

```
1 #include <iostream>
2 int main() {
3     // some random algorithm
4 }
```

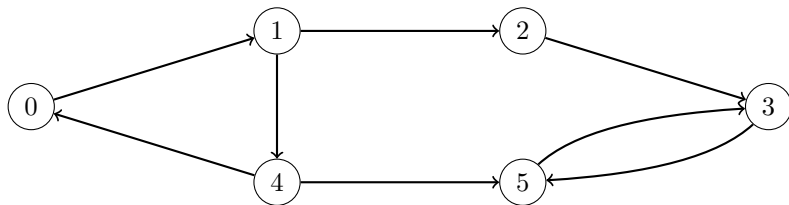
**No!** Take your time,  
model the problem,  
design the algorithm,  
**understand why it should work**,  
⇒ then code.

# 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?")

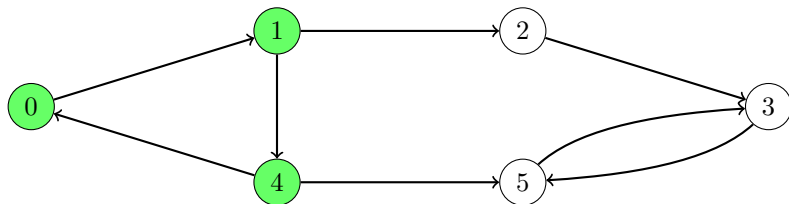
## Tutorial problem: example

What are the universal vertices?



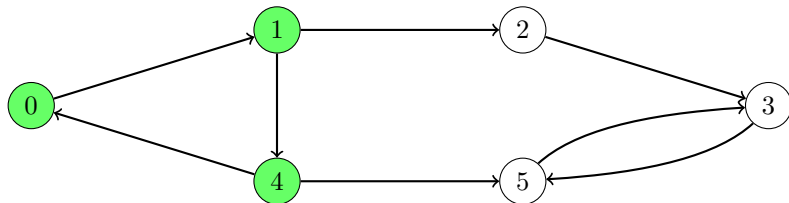
## Tutorial problem: example

What are the universal vertices?



## Tutorial problem: example

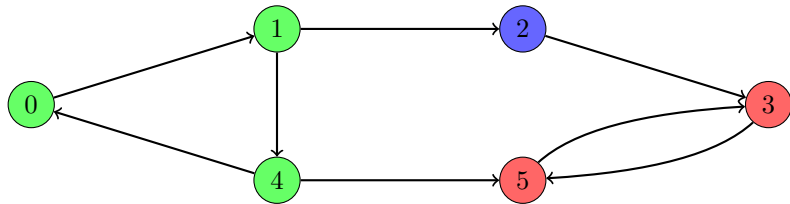
What are the universal vertices?



⇒ must be related to some sort of connected component concept in directed graphs!

## Tutorial problem: strongly connected components (SCC) example

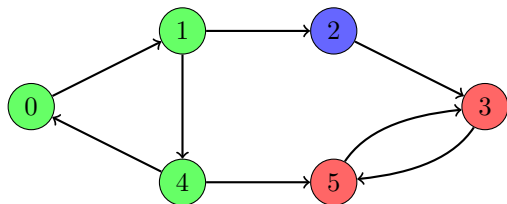
Strongly connected components:





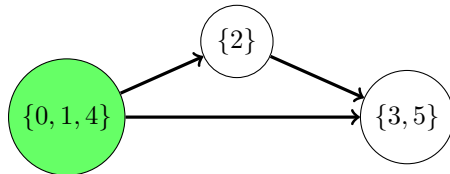
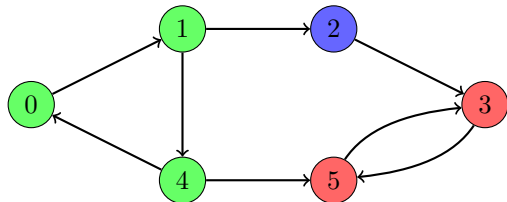
## Tutorial problem: strongly connected components (SCC) example

Strongly connected components:



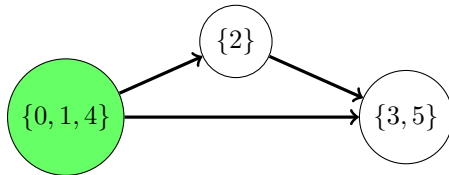
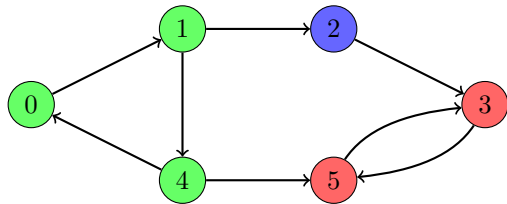
## Tutorial problem: strongly connected components (SCC) example

Strongly connected components:



## Tutorial problem: strongly connected components (SCC) example

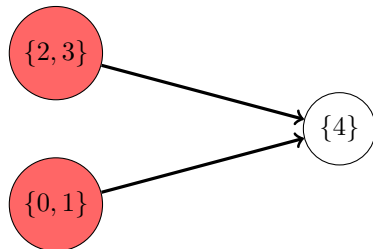
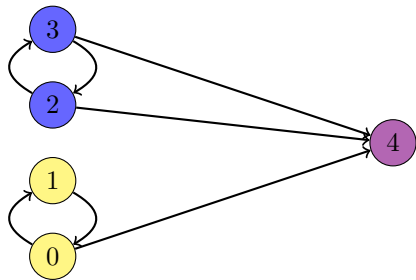
Strongly connected components:



Is there always a universal vertex?

## Tutorial problem: strongly connected components “NO” example

No!



## Tutorial problem: modeling

Let us call a strongly connected component a *minimal component* if it has no in-edges in the strong condensation of the graph (the directed acyclic graph of the strongly connected components).

# Tutorial problem: modeling

Let us call a strongly connected component a *minimal component* if it has no in-edges in the strong condensation of the graph (the directed acyclic graph of the strongly connected components).

## Fact

*If there is more than one minimal component in  $G$ , then there is no universal vertex  $u$ .*

## Lemma

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

i.e the unique minimal component

## 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 best universal vertex  $u$  in  $G$ .

# 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 best universal vertex  $u$  in  $G$ .

Step 1:  $O(|V| + |E|)$  time (DFS)



# 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 best universal vertex  $u$  in  $G$ .

Step 1:  $O(|V| + |E|)$  time (DFS)

Step 2: Dijkstra's algorithm  $\Omega(|V|)$  times  $\Rightarrow \Omega(|V|^2 \log |V| + |V||E|)$  time!  
I.e. around  $|V||E| \approx 10^5 \cdot 2 \cdot 10^5 = 2 \underbrace{0'000'000'000}_{\text{too many zeros}}$  operations.

# Tutorial problem: modeling

Another new formulation of the problem:

1. We work with the **reversed graph**  $G_T$ , where all the edges of  $G$  are reversed.
2. If there exists  $> 1$  **maximal** strongly connected component in  $G_T$ , output NO.  
(maximal component: no out-edge & minimal component: no in-edge)
3. Output the shortest distance  $t \rightarrow u$  for any vertex  $u$  in the unique maximal strongly connected component of  $G_T$ .

Now we can work only with  $G_T$  and one single Dijkstra run!

i.e. around  $|V| \log |V| + |E| \approx 2 \cdot 10^5 = 200'000$  operations.

# Tutorial problem: implementation

How to implement this now?  
First and foremost, [BGL docs](#):

# Tutorial problem: implementation

How to implement this now?

First and foremost, [BGL docs](#):

- ▶ How to find the [strong\\_components](#).

# Tutorial problem: implementation

How to implement this now?

First and foremost, [BGL docs](#):

- ▶ How to find the [strong\\_components](#).
- ▶ How to check how many maximal components are there?  
[topological\\_sort](#)?

Maybe there is a simple ad hoc solution?

# Tutorial problem: implementation

How to implement this now?

First and foremost, [BGL docs](#):

- ▶ How to find the [strong\\_components](#).
- ▶ How to check how many maximal components are there?  
[topological\\_sort](#)?  
Maybe there is a simple ad hoc solution?
- ▶ Compute shortest  $t - u$  path on  $G_T$  with [dijkstra\\_shortest\\_paths](#).

## Tutorial problem: code – preamble

```
10 // STL includes
11 #include <iostream>
12 #include <vector>
13 #include <algorithm>
14 #include <limits>
15 // BGL includes
16 #include <boost/graph/adjacency_list.hpp>
17 #include <boost/graph/strong_components.hpp>
18 #include <boost/graph/dijkstra_shortest_paths.hpp>
```

## Tutorial problem: code – typedefs

```
24 // Directed graph with integer weights on edges.
25 typedef boost::adjacency_list<boost::vecS, boost::vecS, boost::directedS,
26 boost::no_property,
27 boost::property<boost::edge_weight_t, int>
28 >
29           Graph;
29 typedef boost::graph_traits<Graph>::vertex_descriptor    Vertex; // Vertex type
30 typedef boost::graph_traits<Graph>::edge_descriptor      Edge;   // Edge type
31 typedef boost::graph_traits<Graph>::edge_iterator        EdgeIt; // Edge iterator
32 // Property map edge -> weight
33 typedef boost::property_map<Graph, boost::edge_weight_t>::type WeightMap;
```



## Tutorial problem: code – reading the input

```
38 void testcase() {
39     // Read and build graph
40     int V, E, t;           // 1st line: <vertex_no> <edge_no> <target>
41     std::cin >> V >> E >> t;
42     Graph GT(V);           // Creates an empty graph on V vertices
43     WeightMap weightmap = boost::get(boost::edge_weight, GT);
44     for (int i = 0; i < E; ++i) {
45         int u, v, w;       // Each edge: <from> <to> <weight>
46         std::cin >> u >> v >> w;
47         Edge e; bool success;           // *** We swap u and v to create ***
48         boost::tie(e, success) = boost::add_edge(v, u, GT); // *** the reversed graph GT! ***
49         weightmap[e] = w;
50     }
```

## Tutorial problem: code – strong components

```
50 void testcase() {  
51     ...  
52     // Store index of the vertices' strong component; index range [0,nsc)  
53     std::vector<int> sccmap(V); // Use this vector as exterior property map  
54     int nsc = boost::strong_components(GT, // Total number of components  
55     boost::make_iterator_property_map(  
56     sccmap.begin(), boost::get(boost::vertex_index, GT)));
```

## Tutorial problem: code – strong components

```
50 void testcase() {  
51     ...  
52     // Store index of the vertices' strong component; index range [0,nsc)  
53     std::vector<int> sccmap(V); // Use this vector as exterior property map  
54     int nsc = boost::strong_components(GT, // Total number of components  
55     boost::make_iterator_property_map(  
56     sccmap.begin(), boost::get(boost::vertex_index, GT)));
```

**Exterior property:** `strong_components` assigns to each vertex the index of its strong component. This is a *property* of the vertex stored *outside* of the graph itself, namely in the vector `sccmap`. To access the vector, we turn it into an *exterior property map*, i.e., using *`boost::make_iterator_property_map`*.

## Tutorial problem: code – maximal SCCs

```
56 void testcase() {
57     ...
58     // Find universal strong component (if any)
59     // Why does this approach work? Exercise.
60     std::vector<bool> is_max(nsc, true);
61     EdgeIt ebegin, eend;
62     // Iterate over all edges.
63     for (boost::tie(ebegin, eend) = boost::edges(GT); ebegin != eend; ++ebegin) {
64         // ebegin is an iterator, *ebegin is a descriptor.
65         Vertex u = boost::source(*ebegin, GT), v = boost::target(*ebegin, GT);
66         if (sccmap[u] != sccmap[v]) is_max[sccmap[u]] = false;
67         // this edge (u,v) in GT implies that component sccmap[u] is not minimal in G
68     }
69     int max_count = std::count(is_max.begin(), is_max.end(), true);
70     if (max_count != 1) {
71         std::cout << "NO" << std::endl;
72     }
73 }
```

## Tutorial problem: code – Dijkstra

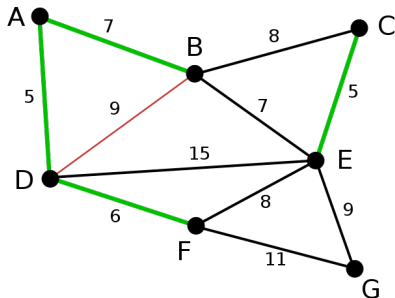
```
73 void testcase() {
74     ...
75     // Compute shortest t-u path in GT
76     std::vector<int> distmap(V);    // We must use at least one of these
77     std::vector<Vertex> predmap(V); // vectors as an exterior property map.
78     boost::dijkstra_shortest_paths(GT, t,
79     predecessor_map(boost::make_iterator_property_map( // named parameters
80     predmap.begin(), boost::get(boost::vertex_index, GT))).
81     distance_map(boost::make_iterator_property_map( // concatenated by .
82     distmap.begin(), boost::get(boost::vertex_index, GT))));
83     int res = std::numeric_limits<int>::max();
84     for (int u = 0; u < V; ++u)
85         // Minimum of distances to 'maximal' universal vertices
86         if (is_max[sccmap[u]])
87             res = std::min(res, distmap[u]);
88     std::cout << res << std::endl;
89 }
```

## Tutorial problem: code – main

```
94 // Main function looping over the testcases
95 int main() {
96     std::ios_base::sync_with_stdio(false);
97     int T;          std::cin >> T;    // First input line: Number of testcases.
98     while(T--)      testcase();
99 }
```

## Recap: Minimum spanning trees

For a connected graph  $G = (V, E)$ , a minimum spanning tree of  $G$  is a subgraph of  $G$  connecting all vertices in  $V$  with the minimum sum of edge weights.



Intermediate step of Kruskal's algorithm to compute a Minimum Spanning Tree.

# Minimum spanning tree implementations

We need to provide an edge vector to Kruskal's algorithm for storing MST edges.

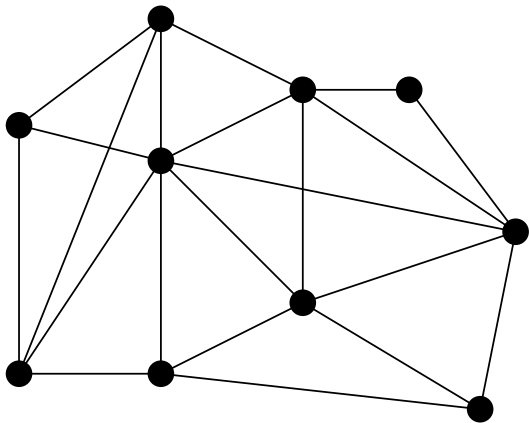
## Kruskal's algorithm

```
1 std::vector<Edge> mst;    // Vector to store MST edges (not a property map!)
2 boost::kruskal_minimum_spanning_tree(G, std::back_inserter(mst));
3 std::vector<Edge>::iterator ebeg, eend = mst.end();
4 for (ebeg = mst.begin(); ebeg != eend; ++ebeg) {
5     ...
```

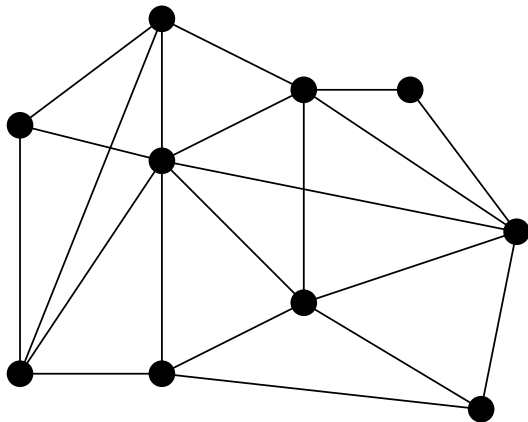


## Recap: Maximum matching

►  $G = (V, E)$

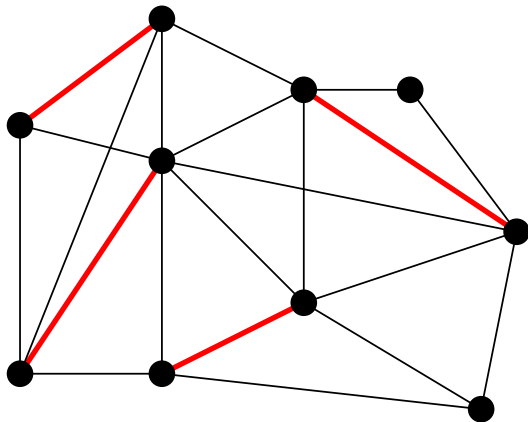


## Recap: Maximum matching



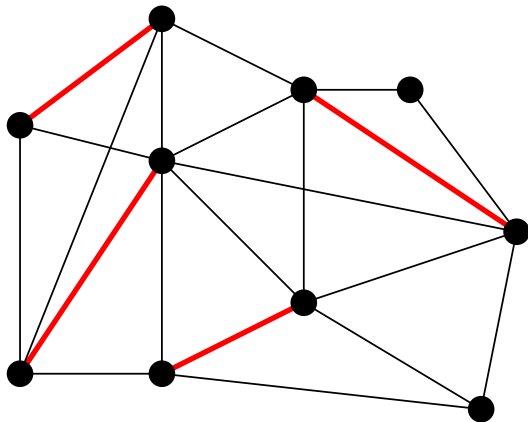
- ▶  $G = (V, E)$
- ▶  $M \subseteq E$  is a matching if and only if no two edges of  $M$  are adjacent.

## Recap: Maximum matching



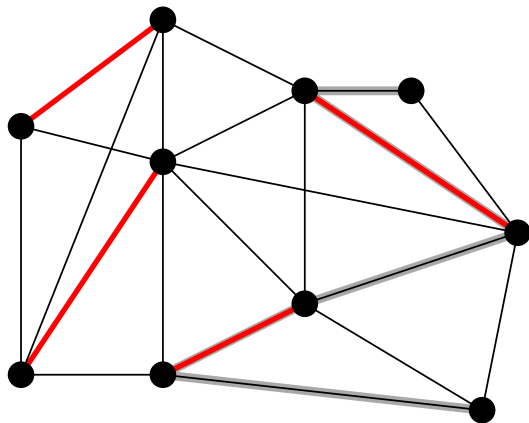
- ▶  $G = (V, E)$
- ▶  $M \subseteq E$  is a matching if and only if no two edges of  $M$  are adjacent.

## Recap: Maximum matching



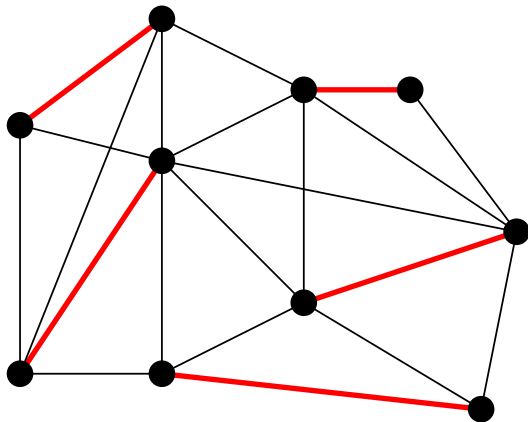
- ▶  $G = (V, E)$
- ▶  $M \subseteq E$  is a matching if and only if no two edges of  $M$  are adjacent.
- ▶ In an unweighted graph, a maximum matching is a matching of maximum cardinality.

## Recap: Maximum matching



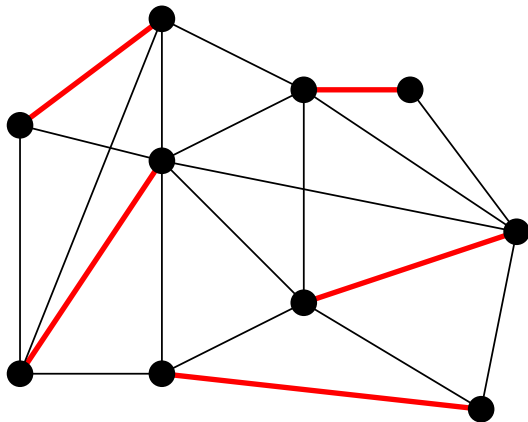
- ▶  $G = (V, E)$
- ▶  $M \subseteq E$  is a matching if and only if no two edges of  $M$  are adjacent.
- ▶ In an unweighted graph, a maximum matching is a matching of maximum cardinality.

## Recap: Maximum matching



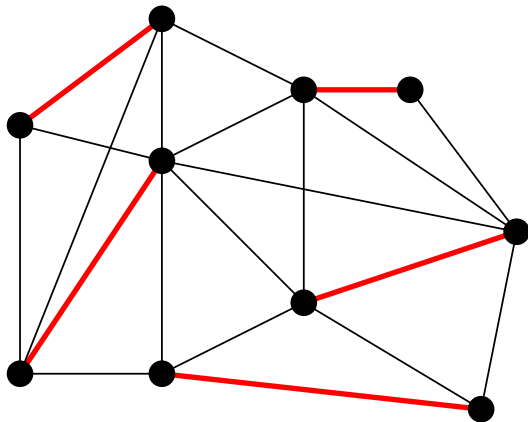
- ▶  $G = (V, E)$
- ▶  $M \subseteq E$  is a matching if and only if no two edges of  $M$  are adjacent.
- ▶ In an unweighted graph, a maximum matching is a matching of maximum cardinality.

## Recap: Maximum matching



- ▶  $G = (V, E)$
- ▶  $M \subseteq E$  is a matching if and only if no two edges of  $M$  are adjacent.
- ▶ In an unweighted graph, a maximum matching is a matching of maximum cardinality.
- ▶ In a weighted graph, a maximum matching is a matching such that the weight sum over the included edges is maximum.

## Recap: Maximum matching



- ▶  $G = (V, E)$
- ▶  $M \subseteq E$  is a matching if and only if no two edges of  $M$  are adjacent.
- ▶ In an unweighted graph, a maximum matching is a matching of maximum cardinality.
- ▶ In a weighted graph, a maximum matching is a matching such that the weight sum over the included edges is maximum.
- ▶ BGL does not provide weighted matching algorithms.



# Maximum matching: invoking algorithm

```
1 // Compute Matching
2 std::vector<Vertex> matemap(V);
3 // Use the vector as an Exterior Property Map: Vertex -> Matched mate
4 boost::edmonds_maximum_cardinality_matching(G, boost::make_iterator_property_map(
5 matemap.begin(), boost::get(boost::vertex_index, G)));
6
7 // Look at the matching
8 // Matching size
9 int matchingsize = boost::matching_size(G, boost::make_iterator_property_map(
10 matemap.begin(), boost::get(boost::vertex_index, G)));
11
12 // unmatched vertices get the NULL_VERTEX as mate.
13 const Vertex NULL_VERTEX = boost::graph_traits<Graph>::null_vertex();
14 for (int i = 0; i < V; ++i) {
15     if (matemap[i] != NULL_VERTEX && i < matemap[i]) {
16     ...
```

## Setup: BGL installation

- ▶ Pre-installed in ETH computer rooms and the Algalab Virtualbox Image.  
Most likely also already installed on your system if you installed CGAL last week.

## Setup: BGL installation

- ▶ Pre-installed in ETH computer rooms and the Algalab Virtualbox Image.  
Most likely also already installed on your system if you installed CGAL last week.
- ▶ On "standard" Linux distributions try getting a package from the repository.  
On macOS package from [Homebrew](#).

## Setup: BGL installation

- ▶ Pre-installed in ETH computer rooms and the Algolab Virtualbox Image.  
Most likely also already installed on your system if you installed CGAL last week.
- ▶ On "standard" Linux distributions try getting a package from the repository.  
On macOS package from [Homebrew](#).
- ▶ Comments on the versions:  
1.61: This version is recommended (current Ubuntu LTS, Algolab VM).  
1.55+: These versions have Mincost-maxflow, should be fine.
- ▶ See the [technical instructions page](#) for more details.

## Setup: BGL without installing

- ▶ BGL is a Header-only library.
- ▶ Download recent version from: <http://www.boost.org/users/download/>.
- ▶ Just unpack the .tar.bz2 file, no installation required, see Section 3 here: [http://www.boost.org/doc/libs/1\\_58\\_0/more/getting\\_started/unix-variants.html](http://www.boost.org/doc/libs/1_58_0/more/getting_started/unix-variants.html).

## Setup: BGL without installing

- ▶ BGL is a Header-only library.
- ▶ Download recent version from: <http://www.boost.org/users/download/>.
- ▶ Just unpack the .tar.bz2 file, no installation required, see Section 3 here: [http://www.boost.org/doc/libs/1\\_58\\_0/more/getting\\_started/unix-variants.html](http://www.boost.org/doc/libs/1_58_0/more/getting_started/unix-variants.html).
- ▶ To build using this version of boost use this command:  

```
g++ -O3 -std=c++11 -I path/to/boost_1_61_0 test.cpp -o test
```

## Setup: BGL without installing

- ▶ BGL is a Header-only library.
- ▶ Download recent version from: <http://www.boost.org/users/download/>.
- ▶ Just unpack the .tar.bz2 file, no installation required, see Section 3 here: [http://www.boost.org/doc/libs/1\\_58\\_0/more/getting\\_started/unix-variants.html](http://www.boost.org/doc/libs/1_58_0/more/getting_started/unix-variants.html).
- ▶ To build using this version of boost use this command:  

```
g++ -O3 -std=c++11 -I path/to/boost_1_61_0 test.cpp -o test
```
- ▶ Explanation: The '-I' flag tells the compiler to include all the files from this directory, so that it can find header files like 'boost/graph/adjacency\_list.hpp'

## Setup: compilation problems

Error messages can be terrible.



## Setup: compilation problems

Error messages can be terrible.

- ▶ Consider re-compiling the code after every line after it is first written. This will help to identify the problem quickly.
- ▶ Especially after the typedefs, and again after building the graph, before you do anything else!

## Setup: compilation problems

Error messages can be terrible.

- ▶ Consider re-compiling the code after every line after it is first written. This will help to identify the problem quickly.
- ▶ Especially after the typedefs, and again after building the graph, before you do anything else!
- ▶ There will be confusing typedefs, nested types, iterators etc. Come up with a naming pattern and stick to it.

## Setup: runtime problems

## Setup: runtime problems

- ▶ 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 mistakes in your edge insertion.

## Setup: Problem of the week

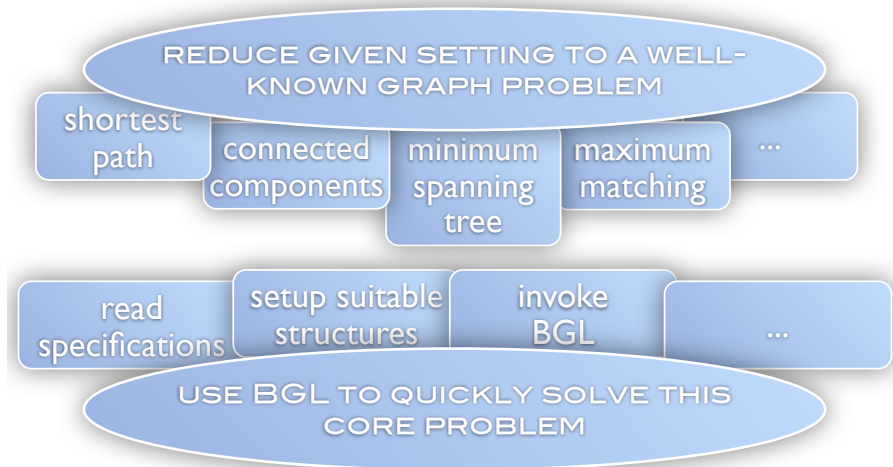
As usual, on Monday. Don't miss it!  
Be advised it doesn't have to be BGL.  
Anything already covered in the course can be used.

## Conclusion



# BGL

THE BOOST GRAPH LIBRARY



# Useful stuff: Algolab BGL documentation

For more information please have a look at the following provided files:

<b>Tutorial slides</b>	A PDF of today's tutorial. Homework: Section Useful stuff.
Copy & paste	A PDF manual containing code snippets and some detailed explanations of the concepts presented in all BGL tutorials.
Tutorial problem	Code and Input file of today's tutorial problem.
<b>Code snippets</b>	Self contained code demonstrating many useful code snippets. Some of it can also be found in the rest of this Section.

# Useful stuff: Options for adjacency\_list

**adjacency\_list** is the class you almost always need.

```
1 // Graph Type, OutEdgeList Type, VertexList Type, (un)directedS
2 typedef adjacency_list<vecS, vecS, undirectedS,
3 no_property,                // nested vertex properties
4 property<edge_weight_t, int> // nested edge properties
5 >
   Graph;
```



# Useful stuff: Options for adjacency\_list

**adjacency\_list** is the class you almost always need.

```
1 // Graph Type, OutEdgeList Type, VertexList Type, (un)directedS
2 typedef adjacency_list<vecS, vecS, undirectedS,
3 no_property,                      // nested vertex properties
4 property<edge_weight_t, int>      // nested edge properties
5 > Graph;
```

**OutEdgeList** (1st vecS) — for each vertex, adjacency list kept in a vector.  
Choosing setS instead disallows parallel edges.

**VertexList** (2nd vecS) — a list of all edges is kept in a vector. Use this!

**Directivity** directedS — directed graph.  
Other choices: undirectedS (undirected graph).  
Rarely needed: bidirectionalS (efficient access to incoming edges)

## Useful stuff: Building a graph

```
1 Graph G(n);    // Constructs empty graph with n vertices
2 ...
3 Edge e;
4 bool success;
5 tie(e, success) = add_edge(u, v, G);
```

## Useful stuff: Building a graph

```
1 Graph G(n);    // Constructs empty graph with n vertices
2 ...
3 Edge e;
4 bool success;
5 tie(e, success) = 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*.

## Useful stuff: Building a graph

```
1 Graph G(n);    // Constructs empty graph with n vertices
2 ...
3 Edge e;
4 bool success;
5 tie(e, success) = 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 a failure (when the edge is a duplicate).

# Useful stuff: Removing vertices and edges, Clearing a graph

**Dangerous:** Deletions of single vertices and edges.

Takes time, invalidates descriptors and iterators, might behave counterintuitively. Consult the docs. Not recommended.

```
1 remove_edge(u, v, G);  
2 remove_edge(e, G);  
3 clear_vertex(u, G);  
4 clear_out_edges(u, G);  
5 remove_vertex(u, G);
```

# Useful stuff: Removing vertices and edges, Clearing a graph

**Dangerous:** Deletions of single vertices and edges.

Takes time, invalidates descriptors and iterators, might behave counterintuitively. Consult the docs. Not recommended.

```
1 remove_edge(u, v, G);  
2 remove_edge(e, G);  
3 clear_vertex(u, G);  
4 clear_out_edges(u, G);  
5 remove_vertex(u, G);
```

**OK:** Clearing a graph once it is no longer needed.

```
1 G.clear(); // Removes all edges and vertices.  
2 G = Graph(n); // Destroys old graph; creates a new one with n vertices.
```

# Useful stuff: Iterators

```
1 // Iterating over vertices
2 for (u = 0; u < num_vertices(G); ++u) {
3   ...
4 // Iterating over edges
5 EdgeIt eit, eend;
6 for (tie(eit, eend) = edges(G); eit != eend; ++eit) {
7   // eit is EdgeIterator, *eit is EdgeDescriptor
8   Vertex u = source(*eit, G), v = target(*eit, G);
9   ...
```

- ▶ `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.

# Useful stuff: Iterators

```
1 // Iterating over vertices
2 for (u = 0; u < num_vertices(G); ++u) {
3   ...
4 // Iterating over edges
5 EdgeIt eit, eend;
6 for (tie(eit, eend) = edges(G); eit != eend; ++eit) {
7 // eit is EdgeIterator, *eit is EdgeDescriptor}
8 Vertex u = source(*eit, G), v = target(*eit, G);
9 ...
```

- ▶ 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.

```
10 // Iterating over outgoing edges
11 OutEdgeIt oeit, oeend;
12 for (tie(oeit, oeend) = out_edges(u, G); oeit != oeend; ++oeit) {
13 Vertex v = target(*oeit, G);
14 ...
```

- ▶ source(\*eit, G) is guaranteed to be u, even in an undirected graph.



## Useful stuff: Interior property maps – vertices

Think of a **property map** as a map (i.e., object with operator `[]`) indexed by vertices or edges. Property maps of vertices could be simulated with a vector, but maps of edges are very convenient.

```
1 // Note the nested syntax for defining more than one vertex property.
2 typedef adjacency_list<vecS, vecS, directedS,
3   property<vertex_name_t, string,
4   property<vertex_distance_t, int> > >      Graph;
5 typedef property_map<Graph, vertex_name_t>::type      NameMap;
6 typedef property_map<Graph, vertex_distance_t>::type  DistMap;
7 ...
8 NameMap namemap = get(vertex_name, G);
9 namemap[u] = "Hans";
```

## Useful stuff: Interior property maps – vertices

Think of a **property map** as a map (i.e., object with operator `[]`) indexed by vertices or edges. Property maps of vertices could be simulated with a vector, but maps of edges are very convenient.

```
1 // Note the nested syntax for defining more than one vertex property.
2 typedef adjacency_list<vecS, vecS, directedS,
3   property<vertex_name_t, string,
4   property<vertex_distance_t, int> > >      Graph;
5 typedef property_map<Graph, vertex_name_t>::type      NameMap;
6 typedef property_map<Graph, vertex_distance_t>::type DistMap;
7 ...
8 NameMap namemap = get(vertex_name, G);
9 namemap[u] = "Hans";
```

- ▶ `namemap` is just a handle (pointer), copying it costs  $\mathcal{O}(1)$ .
- ▶ `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.

## Useful stuff: Interior property maps – edges

```
1 typedef adjacency_list<vecS, vecS, directedS,  
2 no_property, // No vertex properties this time.  
3 // Edge properties as fifth template argument.  
4 property<edge_weight_t, int> > Graph;  
5 typedef property_map<Graph, edge_weight_t>::type WeightMap;  
6 ...  
7 WeightMap weightmap = get(edge_weight, G);  
8 weightmap[e] = cost;
```

- ▶ weightmap is used by many algorithms (Prim, Dijkstra, Kruskal, ...) as default choice for the edge weight.

## Useful stuff: Predefined properties

Some *predefined* vertex and edge properties:

- ▶ `vertex_name_t`
- ▶ `vertex_distance_t`
- ▶ `vertex_color_t`
- ▶ `vertex_degree_t`
- ▶ `edge_name_t`
- ▶ `edge_weight_t`
- ▶ `edge_weight2_t`

Do not be misled into, e.g., thinking that `vertex_degree_t` will automatically keep track of the degree for you.

[More in the source code](#)

## Useful stuff: Custom properties

Can be defined if you want to keep additional info associated with edges.

```
1 namespace boost {  
2   enum edge_info_t { edge_info = 219 }; // A unique ID.  
3   BOOST_INSTALL_PROPERTY(edge, info);  
4 }  
5 struct EdgeInfo {  
6   ...  
7 };  
8 ...  
9 typedef adjacency_list<vecS, vecS, directedS,  
10 no_property,  
11 property<edge_info_t, EdgeInfo> > Graph;  
12 typedef property_map<Graph, edge_info_t>::type InfoMap;  
13 ...  
14 InfoMap infomap = get(edge_info, G);  
15 infomap[e] = ...
```

# Useful stuff: Named parameters I

Using `named parameters` is a way to pass parameters (usually property maps) to functions (BGL algorithms) which is useful in two cases:

# Useful stuff: Named parameters I

Using **named parameters** is a way to pass parameters (usually property maps) to functions (BGL algorithms) which is useful in two cases:

1. Many algorithms have a long list of parameters. Without named parameters, all of these must be provided in the correct order, even if only some are actually needed:

```
1 // Prim non-named parameters example
2 prim_minimum_spanning_tree(G, startvertex,
3 make_iterator_property_map(predmap.begin(), get(vertex_index, G)),
4 make_iterator_property_map(distmap.begin(), get(vertex_index, G)),
5 get(edge_weight, G), get(vertex_index, G), default_dijkstra_visitor());
6 // Prim named parameters:
7 // PredecessorMap must be provided, all other parameters optional
8 prim_minimum_spanning_tree(G,
9 make_iterator_property_map(predmap.begin(), get(vertex_index, G)),
10 root_vertex(startvertex));
```

For e.g. Dijkstra calling the non-named parameter version is even worse!

## Useful stuff: Named parameters II

Using **named parameters** is a way to pass parameters (usually property maps) to functions (BGL algorithms) which is useful in two cases:

2. Some algorithms can record additional information to exterior property maps if provided by named parameters.

```
1 // Kruskal standard example
2 kruskal_minimum_spanning_tree(G, back_inserter(mst));
3 // Kruskal recording Union-Find information
4 vector<int> rankmap(num_vertices(G)); // used by Union-Find
5 vector<Vertex> predmap(num_vertices(G)); // in Union-Find, not the MST!
6 kruskal_minimum_spanning_tree(G, back_inserter(mst),
7 rank_map(make_iterator_property_map(
8 rankmap.begin(), get(vertex_index, G))). // concatenate with .
9 predecessor_map(make_iterator_property_map(
10 predmap.begin(), get(vertex_index, G))));
```

Always concatenate named parameters by a `.`

Do not pass them as separate parameters (i.e. separated by a `,`).



## Useful stuff: Where to be careful

Be careful when you deviate from the provided instructions, in particular if. . .

# Useful stuff: Where to be careful

Be careful when you deviate from the provided instructions, in particular if...

1. ...you use a pointer type as a property map (see e.g. [here](#)):  
Buggy for Dijkstra calls in combination with Strong components header.

```
1 // What we teach (and what works):  
2 dijkstra_shortest_paths(G, 0, distance_map(  
3 make_iterator_property_map(dist.begin(), get(vertex_index, G))));  
4 // Using a pointer type (works most of the time):  
5 dijkstra_shortest_paths(G, 0, distance_map(&dist[0]));
```

# Useful stuff: Where to be careful

Be careful when you deviate from the provided instructions, in particular if...

1. ...you use a pointer type as a property map (see e.g. [here](#)):  
Buggy for Dijkstra calls in combination with Strong components header.

```
1 // What we teach (and what works):  
2 dijkstra_shortest_paths(G, 0, distance_map(  
3 make_iterator_property_map(dist.begin(), get(vertex_index, G))));  
4 // Using a pointer type (works most of the time):  
5 dijkstra_shortest_paths(G, 0, distance_map(&dist[0]));
```

2. ...you use **bundled properties** instead of nested properties:  
Not well documented in the BGL examples; Buggy for MinCost flows.

# Useful stuff: Where to be careful

Be careful when you deviate from the provided instructions, in particular if...

1. ...you use a pointer type as a property map (see e.g. [here](#)):  
Buggy for Dijkstra calls in combination with Strong components header.

```
1 // What we teach (and what works):  
2 dijkstra_shortest_paths(G, 0, distance_map(  
3 make_iterator_property_map(dist.begin(), get(vertex_index, G))));  
4 // Using a pointer type (works most of the time):  
5 dijkstra_shortest_paths(G, 0, distance_map(&dist[0]));
```

2. ...you use **bundled properties** instead of nested properties:  
Not well documented in the BGL examples; Buggy for MinCost flows.
3. ...you use named parameters for flow algorithms: Buggy for MinCost flows.  
Stick to the non-named versions, provide all property maps in correct order.