The Boost Graph Library

The Boost Graph Library

- Header only C++ library

- **GNU/Linux**: (Almost centrainly) already in the repositories of your distribution

```
sudo apt install libboost-graph-dev
```

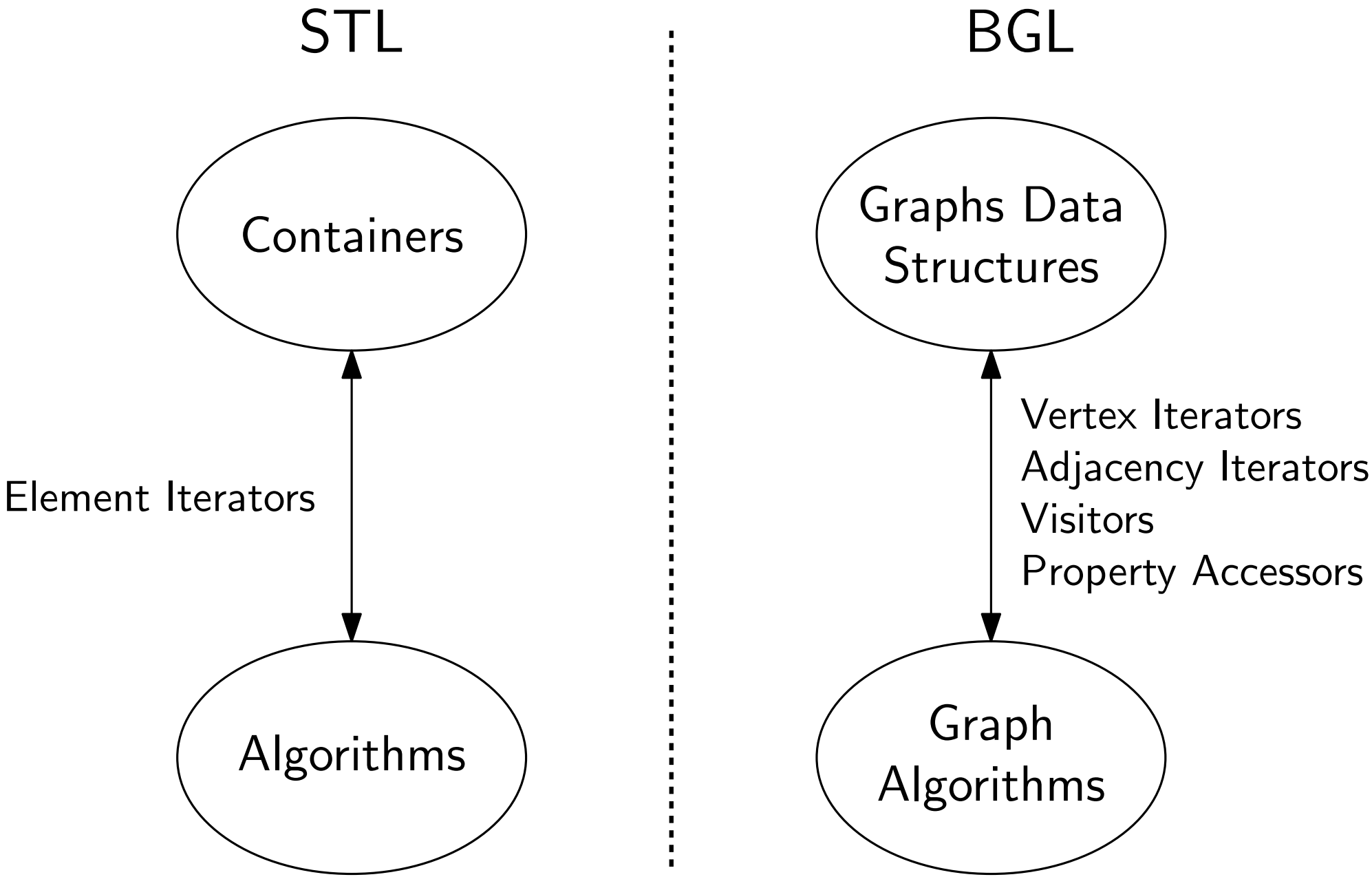
```
sudo pacman -S boost
```

- **Windows:** follow official instructions.

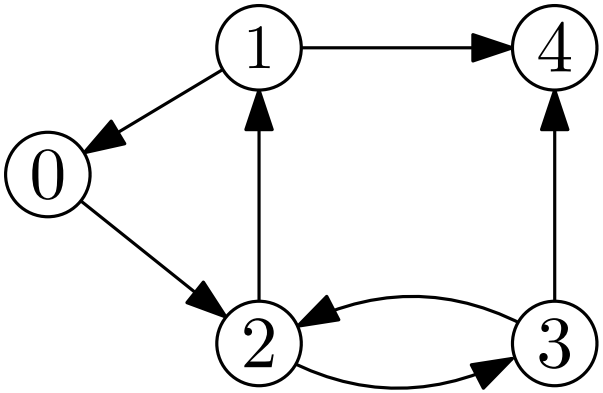
`https://www.boost.org/doc/libs/release/more/getting_started/windows.html`

Essentially: download, unpack, and set the include path of your compiler

The Boost Graph Library



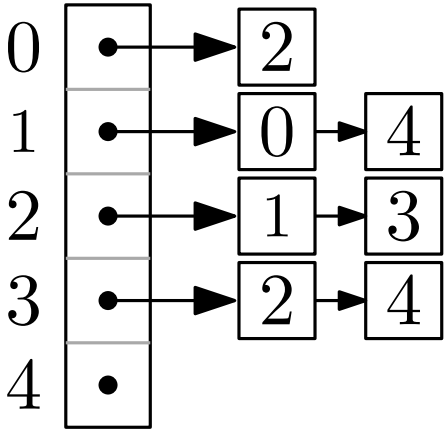
Graph Representation



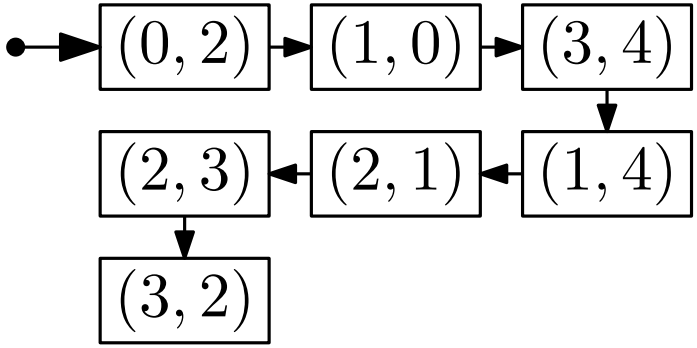
Adjacency Matrix

	0	1	2	3	4
0	0	0	1	0	0
1	1	0	0	0	1
2	0	1	0	1	0
3	0	0	1	0	1
4	0	0	0	0	0

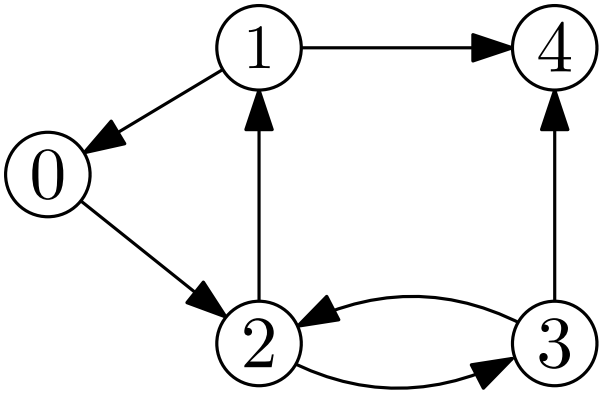
Adjacency List



Edge List



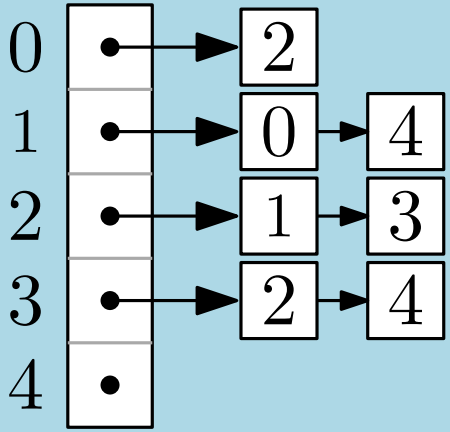
Graph Representation



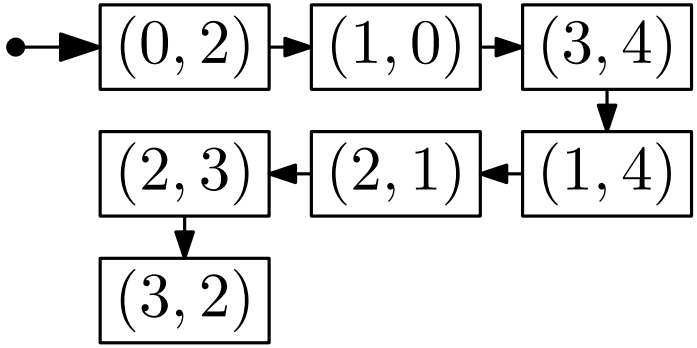
Adjacency Matrix

	0	1	2	3	4
0	0	0	1	0	0
1	1	0	0	0	1
2	0	1	0	1	0
3	0	0	1	0	1
4	0	0	0	0	0

Adjacency List



Edge List



Graph Representations

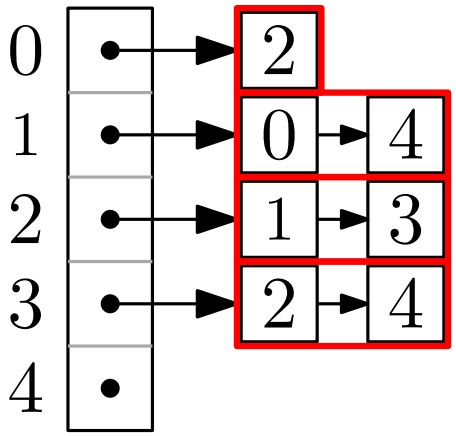
```
#include <boost/graph/adjacency_list.hpp>


// OutEdgeList type, VertexList type, Directivity
typedef boost::adjacency_list<boost::vecS, boost::vecS, boost::directedS> Graph;
```

Graph Representations

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


// OutEdgeList type, VertexList type, Directivity
typedef boost::adjacency_list<boost::vecS, boost::vecS, boost::directedS> Graph;
```

OutEdgeList

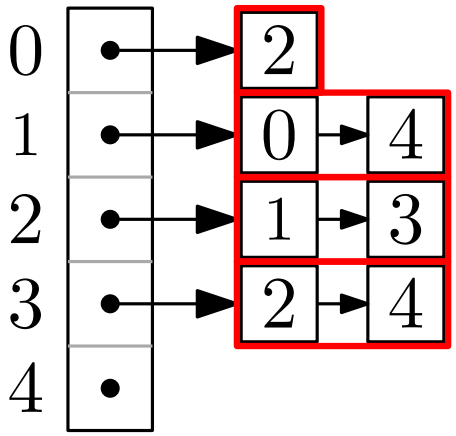
Container used to store the outgoing edges from a vertex



Graph Representations

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


// OutEdgeList type, VertexList type, Directivity
typedef boost::adjacency_list<boost::vecS, boost::vecS, boost::directedS> Graph;
```

OutEdgeList
Container used to store the outgoing edges from a vertex



vecS: vector. Parallel edges allowed. Random acc. iterator. Insertions in amortized $O(1)$ time.

listS: doubly-linked list. Parallel edges allowed. Insertions in constant worst-case time. Space overhead.

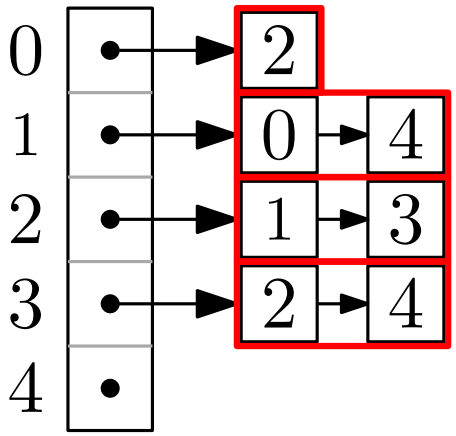
slistS: singly-linked list. Parallel edges allowed. Insertions in constant worst-case time. (Less) space overhead. Cannot iterate backwards.

setS: associative container. No parallel edges. Fast edge lookups, insertions in $O(\log \Delta)$ time.

Graph Representations

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


// OutEdgeList type, VertexList type, Directivity
typedef boost::adjacency_list<boost::vecS, boost::vecS, boost::directedS> Graph;
```

OutEdgeList
Container used to store the outgoing edges from a vertex



vecS: vector. Parallel edges allowed. Random acc. iterator. Insertions in amortized $O(1)$ time.

listS: doubly-linked list. Parallel edges allowed. Inser ce overh

slistS ved. Insertions in constant worst-case time. (Less) space overhead. Cannot iterate backwards.

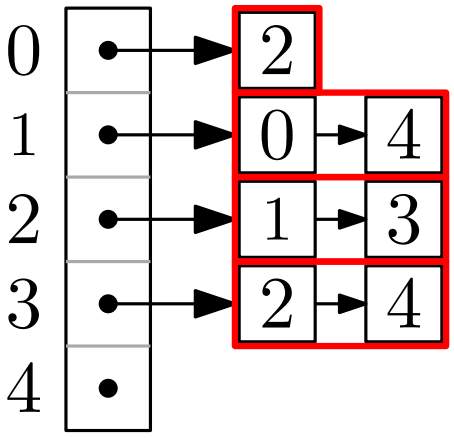
Do I need parallel edges?
Do I need fast edge lookups?

setS: associative container. No parallel edges. Fast edge lookups, insertions in $O(\log \Delta)$ time.

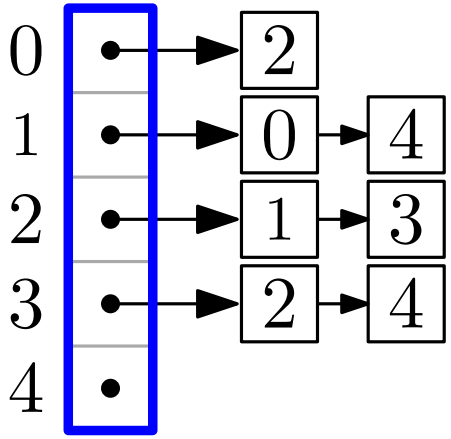
Graph Representations

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


// OutEdgeList type, VertexList type, Directivity
typedef boost::adjacency_list<boost::vecS, boost::vecS, boost::directedS> Graph;
```

OutEdgeList
Container used to store the outgoing edges from a vertex



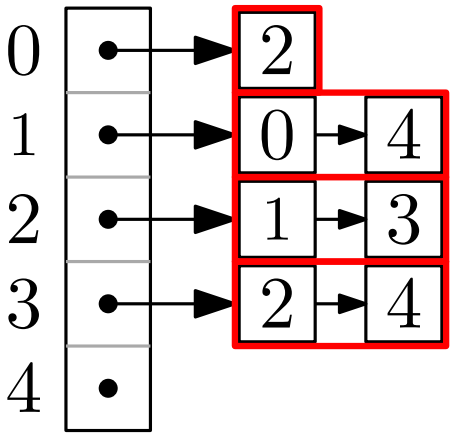
VertexList
Container used to store vertices



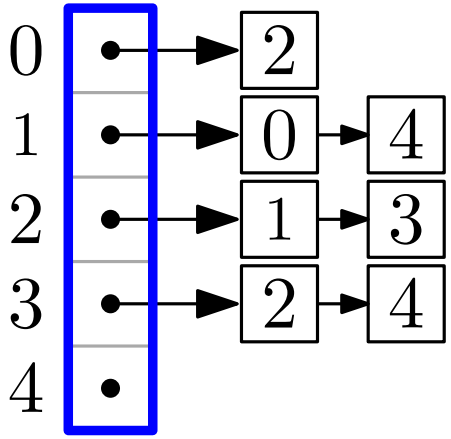
Graph Representations

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


// OutEdgeList type, VertexList type, Directivity
typedef boost::adjacency_list<boost::vecS, boost::vecS, boost::directedS> Graph;
```

OutEdgeList
Container used to store the outgoing edges from a vertex



VertexList
Container used to store vertices



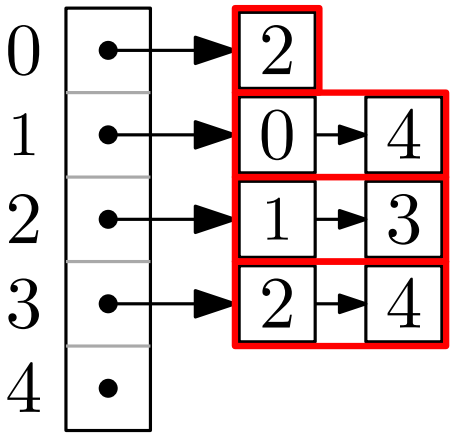
vecS: vector. Amortized $O(1)$-time vertex insertion.

listS: list. Insertions and removals in $O(1)$ worst-case time. Space overhead.

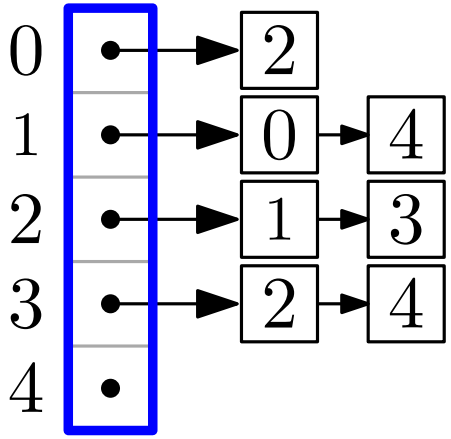
Graph Representations

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


// OutEdgeList type, VertexList type, Directivity
typedef boost::adjacency_list<boost::vecS, boost::vecS, boost::directedS> Graph;
```

OutEdgeList
Container used to store the outgoing edges from a vertex



VertexList
Container used to store vertices



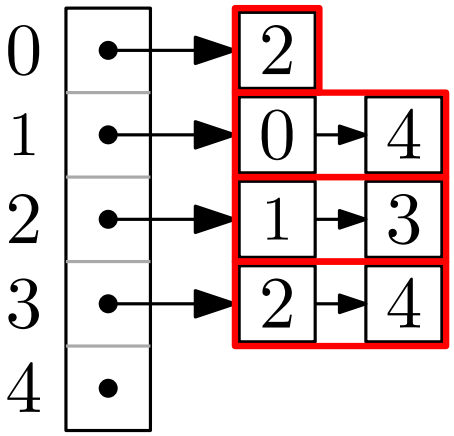
vecS: vector.
Amortized $O(1)$-time vertex insertion.

listS: list. Insertions and removals in $O(1)$ worst-case time.
Space overhead.

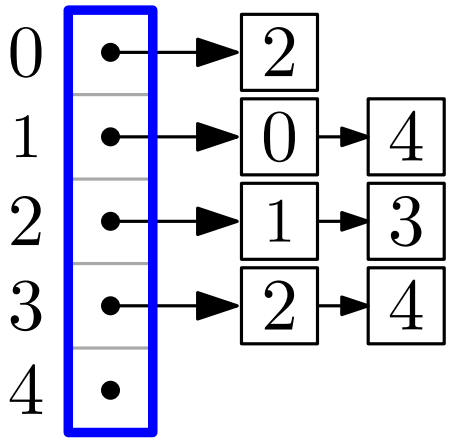
Graph Representations

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


// OutEdgeList type, VertexList type, Directivity
typedef boost::adjacency_list<boost::vecS, boost::vecS, boost::directedS> Graph;
```

OutEdgeList
Container used to store the outgoing edges from a vertex



VertexList
Container used to store vertices



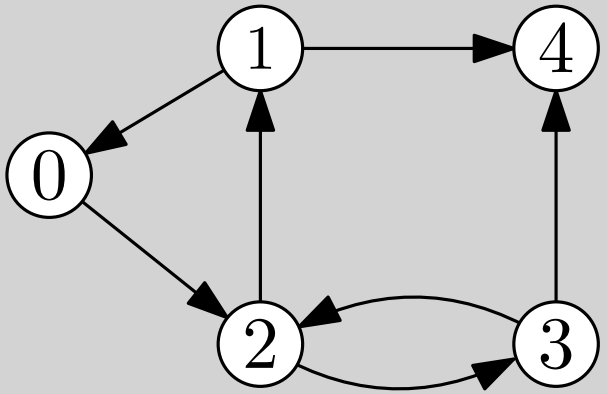
Directivity

- directedS
- undirectedS
- bidirectionalS

(directed with efficient access to incoming edges. Twice the space.)

Constructing a Graph

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

typedef boost::adjacency_list<boost::vecS, boost::vecS,
                              boost::directedS> Graph;

int main()
{
    Graph G(5);

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

    [...]
}
```



Vertex and Edge descriptors

- Vertices and edges are accessed through handles, called vertex descriptors and edge descriptors.
  - They can be, e.g., iterators to the inner collections used by the implementation.
  - Descriptors are **opaque**! You should never explicitly access or manipulate their value!

- `boost::graph_traits<Graph>::edge_descriptor`
- `boost::graph_traits<Graph>::vertex_descriptor`

- **Exception:** If VertexList is of type `vecS`, the vertex descriptor is an integral type that stores indices from 0 to `boost::num_vertices(G)-1`.

Accessing the edges of G

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

typedef boost::adjacency_list<boost::vecS, boost::vecS,
                                                boost::directedS> Graph;
typedef boost::graph_traits<Graph> GT;

int main()
{
    Graph G(5);

    [...]

    std::pair<GT::edge_iterator, GT::edge_iterator> eit;
    for(eit = boost::edges(G); eit.first != eit.second; eit.first++)
    {
        std::cout << "Edge␣from␣" << boost::source(*eit.first, G)
                  << "␣to␣" << boost::target(*eit.first, G) << "\n";
    }

    [...]
}
```

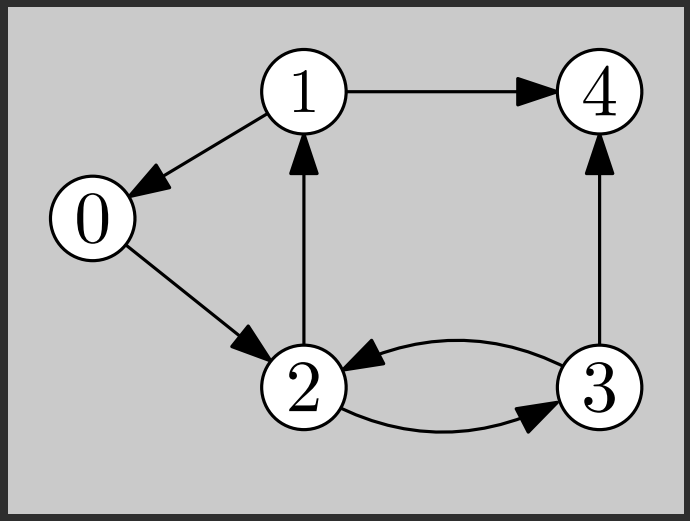
Accessing the edges of G

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

typedef boost::adjacency_list<boost::vecS, boost::vecS,
                                                boost::directedS> Graph;

int main()
{
    Graph G(5);

    [...]

    for(auto [eit, eend] = boost::edges(G); eit != eend; eit++)
    {
        std::cout << "Edge␣from␣" << boost::source(*eit, G)
                  << "␣to␣" << boost::target(*eit, G) << "\n";
    }

    [...]
}
```

Accessing the edges of G

```
#include <boost/graph/adjacency_list.hpp>
#in
typ
                                                              h;
int
{
}
```

```
$ g++ -std=c++17 graph_example.cpp -o graph_example
$
$ ./graph_example
Edge from 0 to 2
Edge from 1 to 0
Edge from 1 to 4
Edge from 2 to 1
Edge from 2 to 3
Edge from 3 to 2
Edge from 3 to 4
$
```



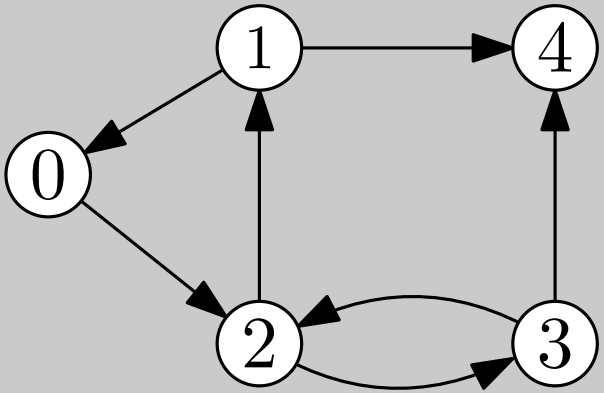
Accessing neighbors/out-edges

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

typedef boost::adjacency_list<boost::vecS, boost::vecS,
                                                boost::directedS> Graph;

int main()
{
    Graph G(5);

    [...]

    //boost::edges(vertex, G) returns a
    //std::pair<GT::out_edge_iterator, GT::out_edge_iterator>
    for(auto [eit, eend] = boost::out_edges(2, G); eit != eend; eit++)
    {
        std::cout << "Out␣edge␣from␣" << boost::source(*eit, G)
                  << "␣to␣" << boost::target(*eit, G) << "\n";
    }

    [...]
}
```

Accessing neighbors/out-edges

```
#include <boost/graph/adjacency_list.hpp>
#in
typ
                                                              h;
int
{
    [...]
}
```

```
$ g++ -std=c++17 graph_example.cpp -o graph_example
$
$ ./graph_example
Out edge from 2 to 1
Out edge from 2 to 3
$
```



Attaching information to edges/vertices

Internal properties:

```
typedef boost::adjacency_list<boost::vecS, boost::vecS, boost::directedS,
        boost::no_property, boost::no_property> Graph;
```

Vertex properties Edge properties

A graph with edge weights

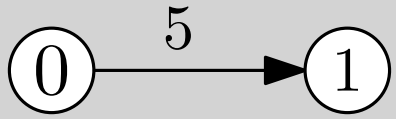
```
typedef boost::adjacency_list<boost::vecS, boost::vecS, boost::directedS,
        boost::no_property, boost::property<boost::edge_weight_t, int>>
        Graph;
```

- `edge_weight_t` is a "property tag": it is one of several predefined property "names", and is used to access the property values.
- `int` is the type of our property: in our example we are going to use integer weights.

Accessing edge/vertex properties

```
typedef boost::adjacency_list<boost::vecS, boost::vecS, boost::directedS,
        boost::no_property, boost::property<boost::edge_weight_t, int>>
        Graph;
typedef boost::graph_traits<Graph> GT;
typedef boost::property_map<Graph, boost::edge_weight_t>::type weight_map;
```

```
int main()
{
  Graph G(2);

  weight_map weights = boost::get(boost::edge_weight, G);

  std::pair<GT::edge_descriptor, bool> e = boost::add_edge(0,1,G);
  weights[e.first] = 5;
}
```



Accessing edge/vertex properties

```
int main()
{
    Graph G(5);
    weight_map weights = boost::get(edge_weight, G);

    [...]

    for(auto [eit, eend] = boost::edges(G); eit != eend; eit++)
    {
        std::cout << "Edge␣from␣" << boost::source(*eit, G)
                  << "␣to␣" << boost::target(*eit, G)
                  << "␣with␣weight␣" << weights[*eit] << "\n";
    }

    [...]
}
```

(Some) BGL Algorithms

Topological Sort

```
void topological_sort(G, result);
```

- `result` is an output iterator.
- Vertices will be written to `result` in **reverse** topological order.

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

Graph G; [...]

std::vector<GT::vertex_descriptor> rev_order;
boost::topological_sort(G, std::back_inserter(rev_order));

for(auto it = rev_order.rbegin(); it!=rev_order.rend(); it++)
    std::cout << *it << "␣";
```

Time: $O(n+m)$

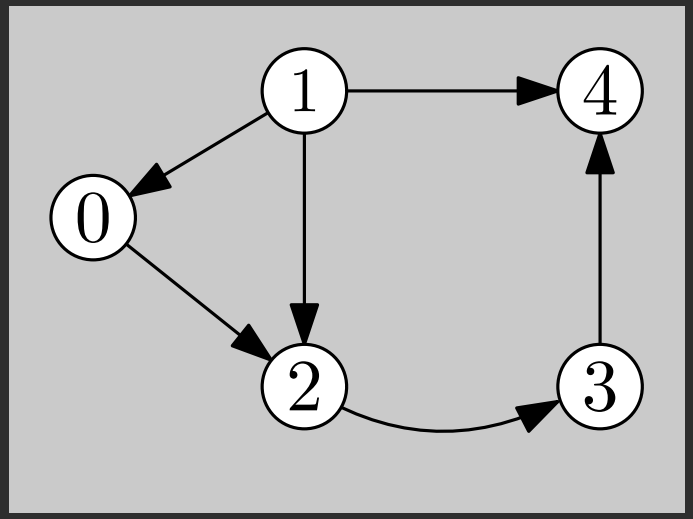
Topological Sort

```
void topological_sort(G, result);
```

-
-

```
$ g++ -std=c++17 ts.cpp -o ts
$
$ ./ts
1 0 2 3 4
$
```

```
#in

Gra

std
boo

for
```



Time: $O(n + m)$

External Properties

It is also possible to store vertex/edge properties in external collections.

Example: Associate an integer to each vertex.

```
std::vector<int> my_prop(boost::num_vertices(G));
```

Tell boost that `my_property` can be indexed with using vertices' indices.

```
make_iterator_property_map(my_prop.begin(), get(boost::vertex_index, G));
```

- The graph has a default interval property with tag `vertex_index_t`.
- It maps vertex descriptors to a vertex index from 0 to `boost::num_vertices(G)-1`.

Connected Components

```
connected_components(G, comp);
```

- Each component will be indexed with a distinct integer.
- The return value is the number of components.
- comp maps each vertex to the index of its component.

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

Graph G; [...]

std::vector<GT::vertices_size_type> comp(boost::num_vertices(G));
boost::connected_components(G,
       make_iterator_property_map(comp.begin(),
           get(boost::vertex_index, G)));

for(int i=0; i<boost::num_vertices(G); i++)
    std::cout << "Vertex " << i
              << " is in component " << comp[i] << "\n";
```

Time: $O(n+m)$

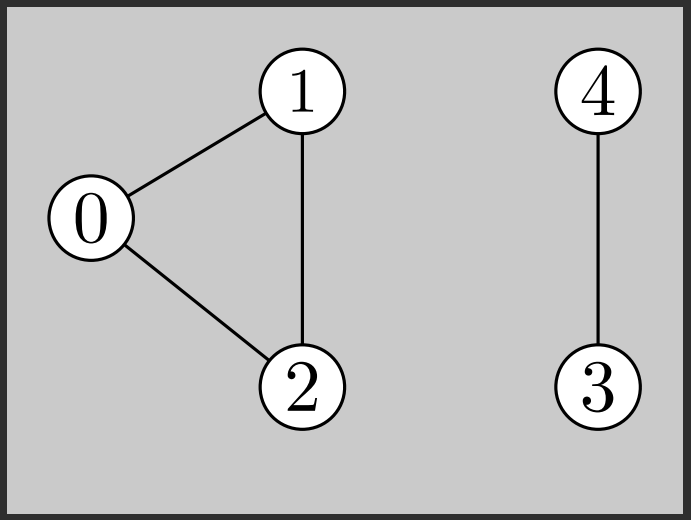
Connected Components

```
connected_components(G, comp);
```

-
-
-

```
#in

Gra

std                                                    ));
boo


for
```

```
$ g++ -std=c++17 cc.cpp -o cc
$
$ ./cc
Vertex 0 is in component 0
Vertex 1 is in component 0
Vertex 2 is in component 0
Vertex 3 is in component 1
Vertex 4 is in component 1
$
```



Time: $O(n+m)$

Strongly Connected Components

```
strong_components(G, comp);
```

- Each component will be indexed with a distinct integer.
- The return value is the number of components.
- comp maps each vertex to the index of its component.

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

Graph G; [...]

std::vector<GT::vertices_size_type> comp(boost::num_vertices(G));
boost::strong_components(G,
       make_iterator_property_map(comp.begin(),
           get(boost::vertex_index, G)));

for(int i=0; i<boost::num_vertices(G); i++)
    std::cout << "Vertex␣" << i
              << "␣is␣in␣component␣" << comp[i] << "\n";
```

Time: $O(n+m)$

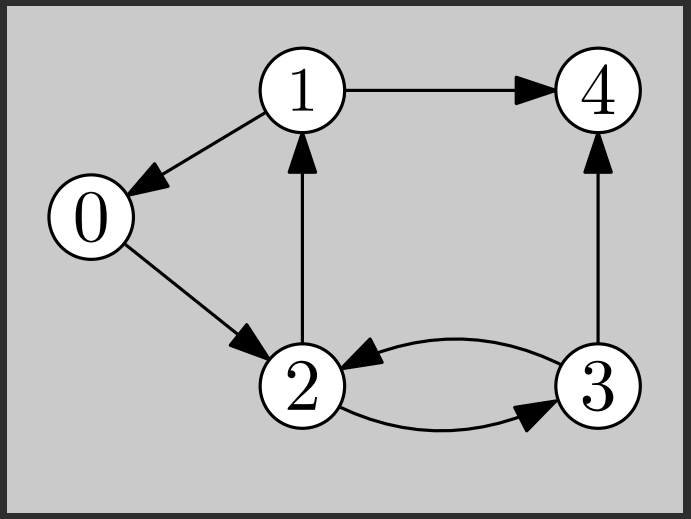
Strongly Connected Components

```
strong_components(G, comp);
```

-
-
-

```
#in

Gra

std                                                    ));
boo


for
```

```
$ g++ -std=c++17 scc.cpp -o scc
$
$ ./scc
Vertex 0 is in component 1
Vertex 1 is in component 1
Vertex 2 is in component 1
Vertex 3 is in component 1
Vertex 4 is in component 0
$
```



Time: $O(n+m)$

Named parameters

To avoid long list of positional parameters to functions, BGL uses *named parameters*.

It uses wrapper classes to match arguments to parameters based on names

Arguments can be passed in any order

Arguments are concatenated using a dot `.` instead of a comma (formally this is a call to a method).

A made-up example:

```
graph_algo(G, root_vertex(v).color(c).weight(w));
```

Strongly Connected Components

```
strong_components(G, comp, named_params...);
```

- One of the named parameters `strong_components` is `root_map`
- For each SCC C, the algorithm picks a unique representative vertex $v_C \in C$

Named parameters:

- `root_map` maps each vertex u of G to the representative vertex v_C of the SCC C containing u.

Strongly Connected Components

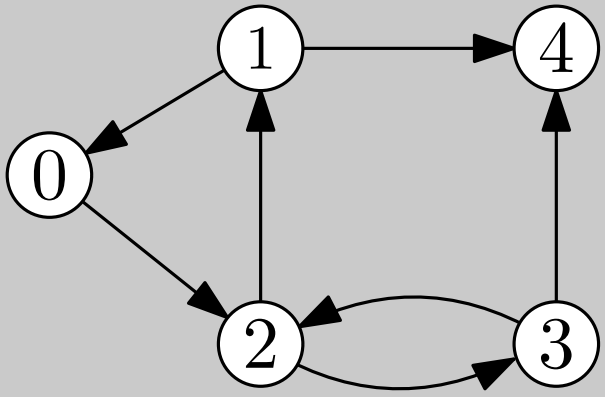
```
strong_components(G, comp, named_params...);
```

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

Graph G; [...]

std::vector<GT::vertices_size_type> comp(boost::num_vertices(G));
std::vector<GT::vertex_descriptor> roots(boost::num_vertices(G));

boost::strong_components(G,
       make_iterator_property_map(comp.begin(),
           get(boost::vertex_index, G)),
       root_map(make_iterator_property_map(roots.begin(),
           get(boost::vertex_index, G))));

for(int i=0; i<boost::num_vertices(G); i++)
   std::cout << "The␣representative␣of␣the␣SCC␣of␣vertex␣" << i
             << "␣is␣" << roots[i] << "\n";
```

Strongly Connected Components

```
strong_components(G, comp, named_params...);
```

```
$ g++ -std=c++17 scc_named.cpp -o scc_named
$
$ ./scc_named
The representative of the SCC of vertex 0 is 0
The representative of the SCC of vertex 1 is 0
The representative of the SCC of vertex 2 is 0
The representative of the SCC of vertex 3 is 0
The representative of the SCC of vertex 4 is 4
```



Example: 2-SAT

$\phi = (x_1 \vee \overline{x_2}) \wedge (x_2 \vee x_4) \wedge (\overline{x_1} \vee \overline{x_3}) \wedge (x_3 \vee x_4) \wedge (\overline{x_1} \vee \overline{x_4})$

Input:

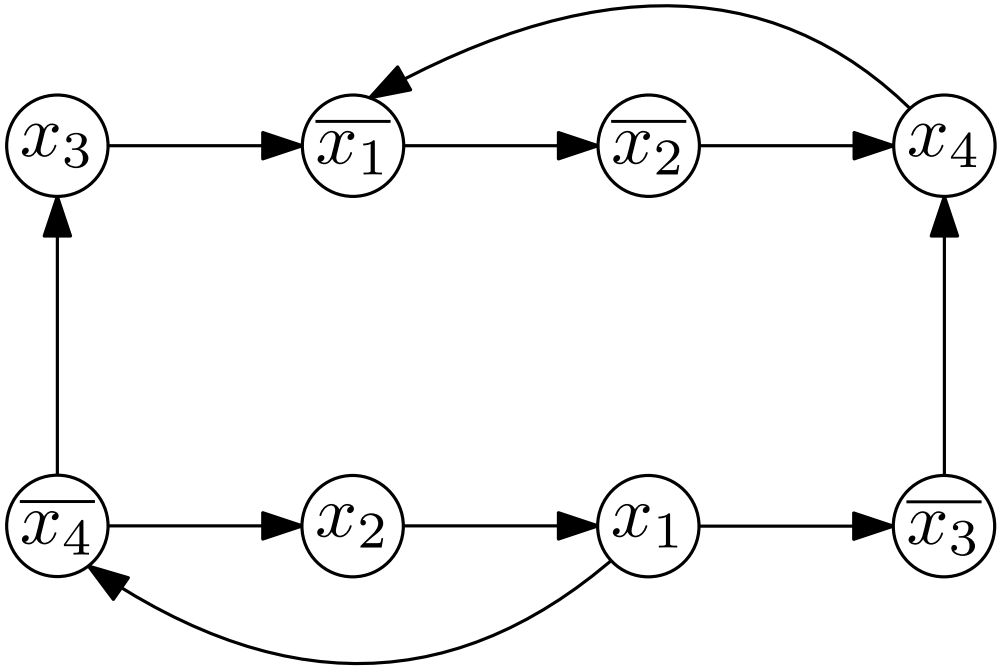
- `n`: number of variables, `m`: number of clauses
- m pairs (i, j) with $i, j \neq 0$.
  A pair (i, j) represents a clause $(a \vee b)$. If $i > 0$ then $a = x_i$.
  If $i < 0$ then $a = \overline{x_{-i}}$. b is definited similarly from j.

```
4  5
 1 -2
 2  4
-1 -3
 3 4
-1 -4
```

Output: True if ϕ is satisfiable. Otherwise false.

Example: 2-SAT

Corollary (from last lecture): ϕ is satisfiable iff $\forall x_i$, x_i and $\overline{x_i}$ belong to different SCCs of the implication graph G_ϕ.



$$\phi = (x_1 \vee \overline{x_2}) \wedge (x_2 \vee x_4) \wedge (\overline{x_1} \vee \overline{x_3}) \wedge (x_3 \vee x_4) \wedge (\overline{x_1} \vee \overline{x_4})$$

Example: 2-SAT

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

//Maps each literal to an id between 0 and 2n-1
int to_vertex_id(int i){ return (i>0)?(2*i-2):(-2*i-1); }

int solve()
{
    int n,m;
    std::cin >> n >> m;

    Graph G(2*n);
    for(int i=0; i<m; i++)
    {
        int x, y;
        std::cin >> x >> y;

        boost::add_edge(to_vertex_id(-x), to_vertex_id(y), G);
        boost::add_edge(to_vertex_id(-y), to_vertex_id(x), G);
    }
```

Example: 2-SAT

```
    std::vector<int> comp(boost::num_vertices(G));
    boost::strong_components(G,
        boost::make_iterator_property_map(comp.begin(),
            get(boost::vertex_index, G)));

    for(int i=1; i<=n; i++)
        if(comp[to_vertex_id(i)] == comp[to_vertex_id(-i)])
            return false;

    return true;
}

int main()
{
    std::cout << (solve()?"Satisfiable\n":"Not␣satisfiable\n");

    return EXIT_SUCCESS;
}
```

Single Source Distances: Dijkstra

```
void dijkstra_shortest_paths(G, v, named_params...);
```

- Runs Dijkstra's algorithm from vertex v.

Named parameters:

- `weight_map` (input): maps edges to their weight. Default: internal map with tag `boost::edge_weight_t`
- `distance_map` (output). maps each vertex with its distance from v. `boost::edge_weight_t`
- `predecessor_map` (output). maps each vertex u to (the vertex descriptor of) its parent p_u in the shortest path tree from v. If u is unreachable from v, or $u = v$, sets $p_u = u$.

Time: $O(m + n \log n)$

Single Source Distances: Dijkstra

```
void dijkstra_shortest_paths(G, v, named_params...);
```

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

Graph G; [...]

std::vector<int> dist(boost::num_vertices(G));
std::vector<GT::vertex_descriptor> pred(boost::num_vertices(G));

boost::dijkstra_shortest_paths(G, 0,
       distance_map(make_iterator_property_map(dist.begin(),
           get(boost::vertex_index, G)))
       .predecessor_map(make_iterator_property_map(pred.begin(),
           get(boost::vertex_index, G))));

for(int i=0; i<boost::num_vertices(G); i++)
    std::cout << "Distance␣to␣" << i << ":␣" << dist[i]
              << "␣Parent:␣" << pred[i] << "\n";
```

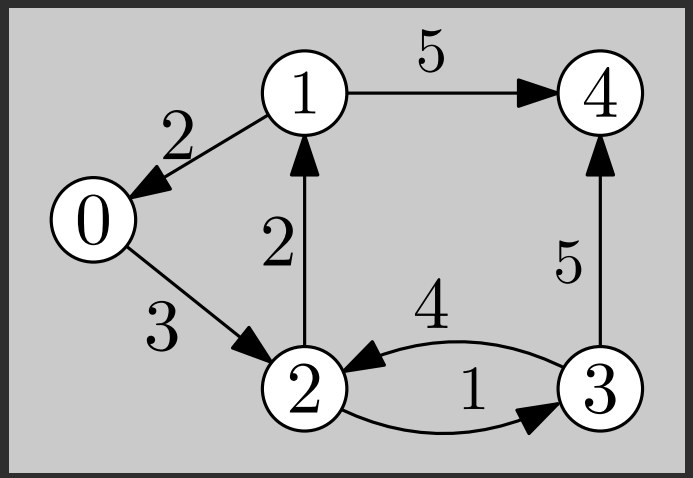
Time: $O(m + n \log n)$

Single Source Distances: Dijkstra

```
void dijkstra_shortest_paths(G, v, named_params...);
```

```
#in
Gra
std
std
boo
for
```

```
$ g++ -std=c++17 dijkstra.cpp -o dijkstra
$
$ ./dijkstra
Distance to 0: 0 Parent: 0
Distance to 1: 5 Parent: 2
Distance to 2: 3 Parent: 0
Distance to 3: 4 Parent: 2
Distance to 4: 9 Parent: 3
```



Time: $O(m + n \log n)$

Single Source Distances in DAGs

```
void dag_shortest_paths(G, v, named_params...);
```

- Compute the the distances / SPT from v in a directed acyclic graph.

Named parameters:

- `weight_map` (input): maps edges to their weight. Default: internal map with tag `boost::edge_weight_t`
- `distance_map` (output). maps each vertex with its distance from v. `boost::edge_weight_t`
- `predecessor_map` (output). maps each vertex u to (the vertex descriptor of) its parent p_u in the shortest path tree from v. If u is unreachable from v, or $u = v$, sets $p_u = u$.

Time: $O(m + n)$

Single Source Distances in DAGs

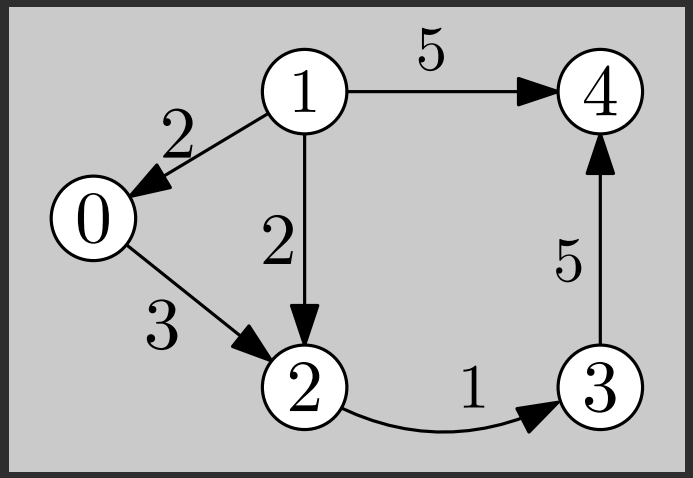
```
void dag_shortest_paths(G, v, named_params...);
```

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

Graph G; [...]

std::vector<int> dist(boost::num_vertices(G));
std::vector<GT::vertex_descriptor> pred(boost::num_vertices(G));

boost::dag_shortest_paths(G, 0,
       distance_map(make_iterator_property_map(dist.begin(),
           get(vertex_index, G))).
       predecessor_map(make_iterator_property_map(pred.begin(),
           get(vertex_index, G)));

for(int i=0; i<boost::num_vertices(G); i++)
    std::cout << "Distance␣to␣" << i << ":␣" << dist[i]
              << "␣Parent:␣" << pred[i] << "\n";
```

Time: $O(m+n)$

Single Source Distances in DAGs

```
$ g++ -std=c++17 dag_sp.cpp -o dag_sp
$
$ ./dag_sp
Distance to 0: 0 Parent: 0
Distance to 1: 2147483647 Parent: 1
Distance to 2: 3 Parent: 0
Distance to 3: 4 Parent: 2
Distance to 4: 9 Parent: 3
```



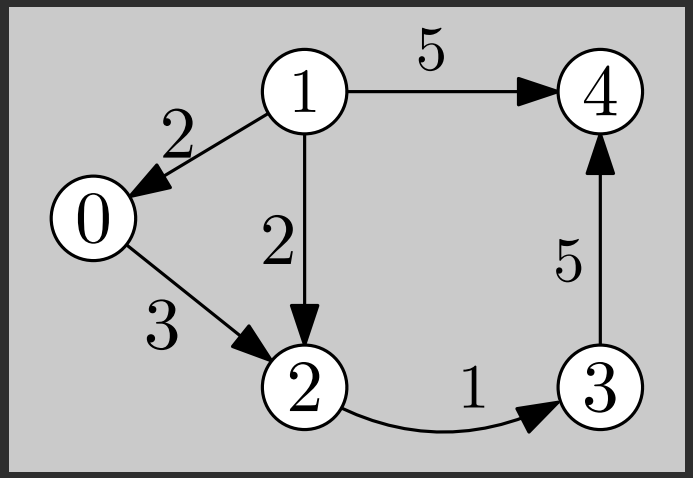
Time: $O(m+n)$

Single Source Distances in DAGs

```
$ g++ -std=c++17 dag_sp.cpp -o dag_sp
$
$ ./dag_sp
Distance to 0: 0 Parent: 0
Distance to 1: 2147483647 Parent: 1
Distance to 2: 3 Parent: 0
Distance to 3: 4 Parent: 2
Distance to 4: 9 Parent: 3
```

unreachable

std::numeric_limits<int>::max()



Time: $O(m+n)$

Single Source Distances: Bellman-Ford

```
bool bellman_ford_shortest_paths(G, named_params...);
```

- Runs Bellman-Ford's algorithm.
- Returns `true` iff all distance from s are finite (no negative cycle found).

Named parameters:

- `root_vertex` (input). The source vertex.
- `weight_map` (input): maps edges to their weight. Default: internal map with tag `boost::edge_weight_t`
- `distance_map` (output). maps each vertex with its distance from v. `boost::edge_weight_t`
- `predecessor_map` (output). maps each vertex u to (the vertex descriptor of) its parent p_u in the shortest path tree from v. If u is unreachable from v, or $u = v$, sets $p_u = u$.

Time: $O(nm)$

Single Source Distances: Bellman-Ford

```
bool bellman_ford_shortest_paths(G, named_params...);
```

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

Graph G; [...]

std::vector<int> dist(boost::num_vertices(G));
std::vector<GT::vertex_descriptor> pred(boost::num_vertices(G));

boost::bellman_ford_shortest_paths(G, root_vertex(0)
       .distance_map(make_iterator_property_map(dist.begin(),
          get(vertex_index, G)))
       .predecessor_map(make_iterator_property_map(pred.begin(),
          get(vertex_index, G)));

for(int i=0; i<boost::num_vertices(G); i++)
    std::cout << "Distance␣to␣" << i << ":␣" << dist[i]
              << "␣Parent:␣" << pred[i] << "\n";
```

Time: $O(nm)$

All to All Distances: Floyd-Warshall

```
bool floyd_warshall_all_pairs_shortest_paths(G, matrix, named_params...);
```

- Runs Floyd-Warshall's algorithm.
- Returns `true` iff there is no negative-weight cycle in G.
- `matrix` stores the computed distance matrix of G.
  `matrix[i][j]` must return a mutable reference, where `i` and `j` are vertex descriptors.

Named parameters:

- `weight_map` (input): maps edges to their weight. Default: internal map with tag `boost::edge_weight_t`

Time: $O(n^3)$

All to All Distances: Floyd-Warshall

```
bool floyd_warshall_all_pairs_shortest_paths(G, matrix, named_params...);
```

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

Graph G; [...]

// n x n matrix of int
std::vector<std::vector<int>> M(boost::num_vertices(G),
                                std::vector<int>(boost::num_vertices(G)));

boost::floyd_warshall_all_pairs_shortest_paths(G, M);

for(auto row : M)
{
    for(int d : row)
        std::cout << d << "␣";
    std::cout << "\n";
}
```

Time: $O(n^3)$

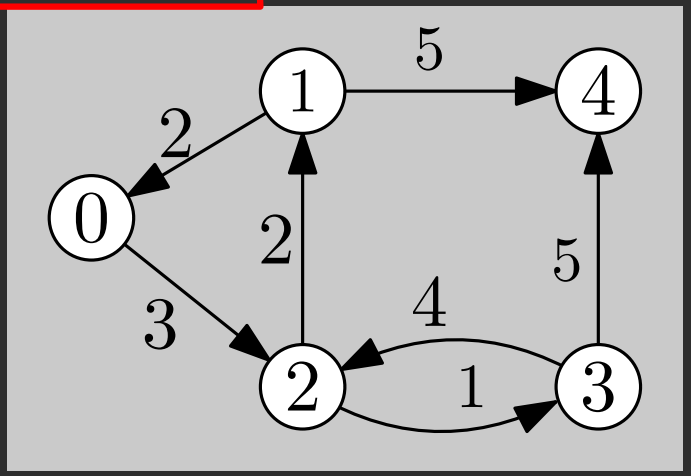
All to All Distances: Floyd-Warshall

```
bool floyd_warshall_all_pairs_shortest_paths(G, matrix, named_params...);
```

```
#in
Gra
//
std
boo
for
{
}
```

```
$ g++ -std=c++17 floyd_warshall.cpp -o floyd_warshall
$
$ ./floyd_warshall.cpp
0 5 3 4 9
2 0 5 6 5
4 2 0 1 6
8 6 4 0 5
2147483647 2147483647 2147483647 2147483647 0
```

```
std::numeric_limits<int>::max()
```



Time: $O(n^3)$

Minimum Spanning Trees: Kruskal

```
kruskal_minimum_spanning_tree(G, result, named_params...);
```

- Runs Kruskal's MST algorithm.
- `result` is an output iterator.
- Edges of the MST are written to `result`.

Named parameters:

- `weight_map` (input): maps edges to their weight. Default: internal map with tag `boost::edge_weight_t`

Time: $O(m \log n)$

Minimum Spanning Trees: Kruskal

```
kruskal_minimum_spanning_tree(G, result, named_params...);
```

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

Graph G; [...]

std::vector<GT::edge_descriptor> edges;

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

for(GT::edge_descriptor e : edges)
        std::cout << "Edge␣from␣" << boost::source(e, G)
                  << "␣to␣" << boost::target(e, G) << "\n";
```

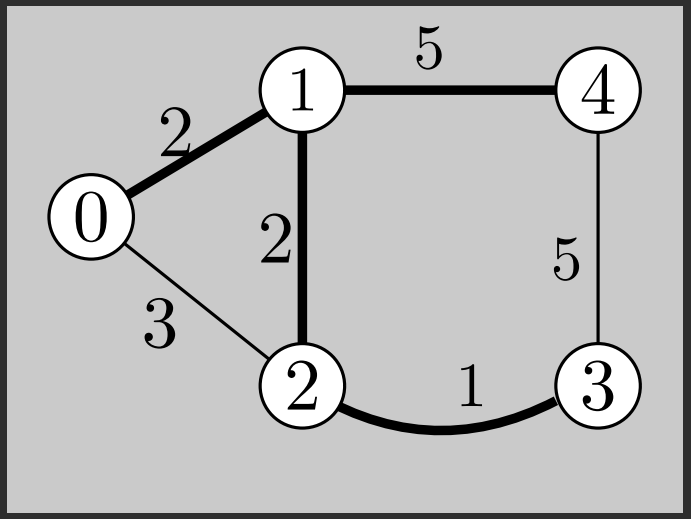
Time: $O(m \log n)$

Minimum Spanning Trees: Kruskal

```
kruskal_minimum_spanning_tree(G, result, named_params...);
```

```
#in
Gra
std
boo                                                    es)
for
```

```
$ g++ -std=c++17 kruskal.cpp -o kruskal
$
$ ./kruskal
Edge from 2 to 3
Edge from 1 to 2
Edge from 1 to 0
Edge from 1 to 4
$
```



Time: $O(m \log n)$

Minimum Spanning Trees: Prim

```
prim_minimum_spanning_tree(G, pred, named_params...);
```

- Runs Prim's MST algorithm.
- `pred` (output). maps each vertex u to (the vertex descriptor of) its parent p_u in the (rooted) MST. If u is the root or is unreachable then $p_u = u$.

Named parameters:

- `weight_map` (input): maps edges to their weight. Default: internal map with tag `boost::edge_weight_t`
- `root_vertex` (input). The root vertex of the MST. Default: the first vertex.

Time: $O(m \log n)$

Minimum Spanning Trees: Prim

```
prim_minimum_spanning_tree(G, pred, named_params...);
```

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

Graph G; [...]

std::vector<GT::vertex_descriptor> pred(boost::num_vertices(G));

boost::prim_minimum_spanning_tree(G,
    make_iterator_property_map(pred.begin(),
        get(boost::vertex_index, G)),
    boost::root_vertex(0));

for(int i=0; i<boost::num_vertices(G); i++)
        std::cout << "Parent␣of␣" << i << ":␣"
                  << pred[i] << "\n";
```

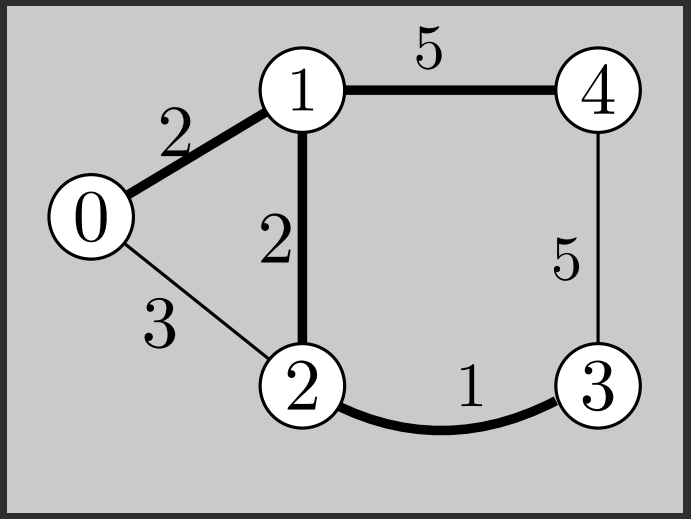
Time: $O(m \log n)$

Minimum Spanning Trees: Prim

```
prim_minimum_spanning_tree(G, pred, named_params...);
```

```
#in
Gra
std                                                    );
boo
for
```

```
$ g++ -std=c++17 prim.cpp -o prim
$
$ ./prim
Parent of 0: 0
Parent of 1: 0
Parent of 2: 1
Parent of 3: 2
Parent of 4: 1
$
```



Time: $O(m \log n)$

Breadth First Search

```
breadth_first_search(G, r, bfs_visitor);
```

- Performs a BFS visit of the graph from vertex r.
- Generates "events" during the visit.
- Events are dispatched by invoking methods of a custom `bfs_visitor` object.

Some events:

- `discover_vertex(v, G)`: A new vertex `v` is discovered.
- `examine_vertex(v, G)`: Vertex `v` is visited.
- `examine_edge(e, G)`: An edge `e` is examined.
- `finish_vertex(v,G)`: All the outgoing edges from `v` have been examined.
- `tree_edge(e, G)`: An edge `e` of the BFS tree is discovered.
- `non_tree_edge(e, G)`: An edge `e` that is *not* in the BFS tree is discovered.

Time: $O(m+n)$

A Custom BFS Visitor

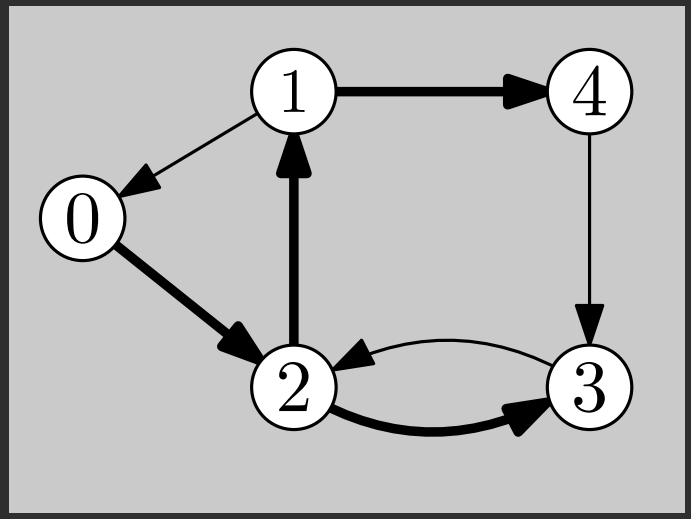
```
#include <boost/graph/breadth_first_search.hpp>

class custom_visitor : public boost::default_bfs_visitor
{
public:
    void examine_vertex(GT::vertex_descriptor u, const Graph& G)
    {
        std::cout << "Visiting␣vertex␣" << u << "\n";
    }

    void tree_edge(GT::edge_descriptor e, const Graph& G)
    {
        std::cout << "Discovered␣tree␣edge␣from␣"
                  << boost::source(e, G) << "␣to␣"
                  << boost::target(e, G) << "\n";
    }
};
```

Starting the BFS Visit

```
Graph G; [...]

//Create an instance of our custom visitor
custom_visitor vis;

//Start the BFS visit
boost::breadth_first_search(G, 0, boost::visitor(vis));
```

Starting the BFS Visit

```
$ g++ -std=c++17 bfs.cpp -o bfs
$
$ ./bfs
Visiting vertex 0
Discovered tree edge from 0 to 2
Visiting vertex 2
Discovered tree edge from 2 to 1
Discovered tree edge from 2 to 3
Visiting vertex 1
Discovered tree edge from 1 to 4
Visiting vertex 3
Visiting vertex 4
$
```



Depth First Search

```
depth_first_search(G, named_params...);
```

- Performs a DFS visit of the graph.
- Generates "events" during the visit.

Named parameters:

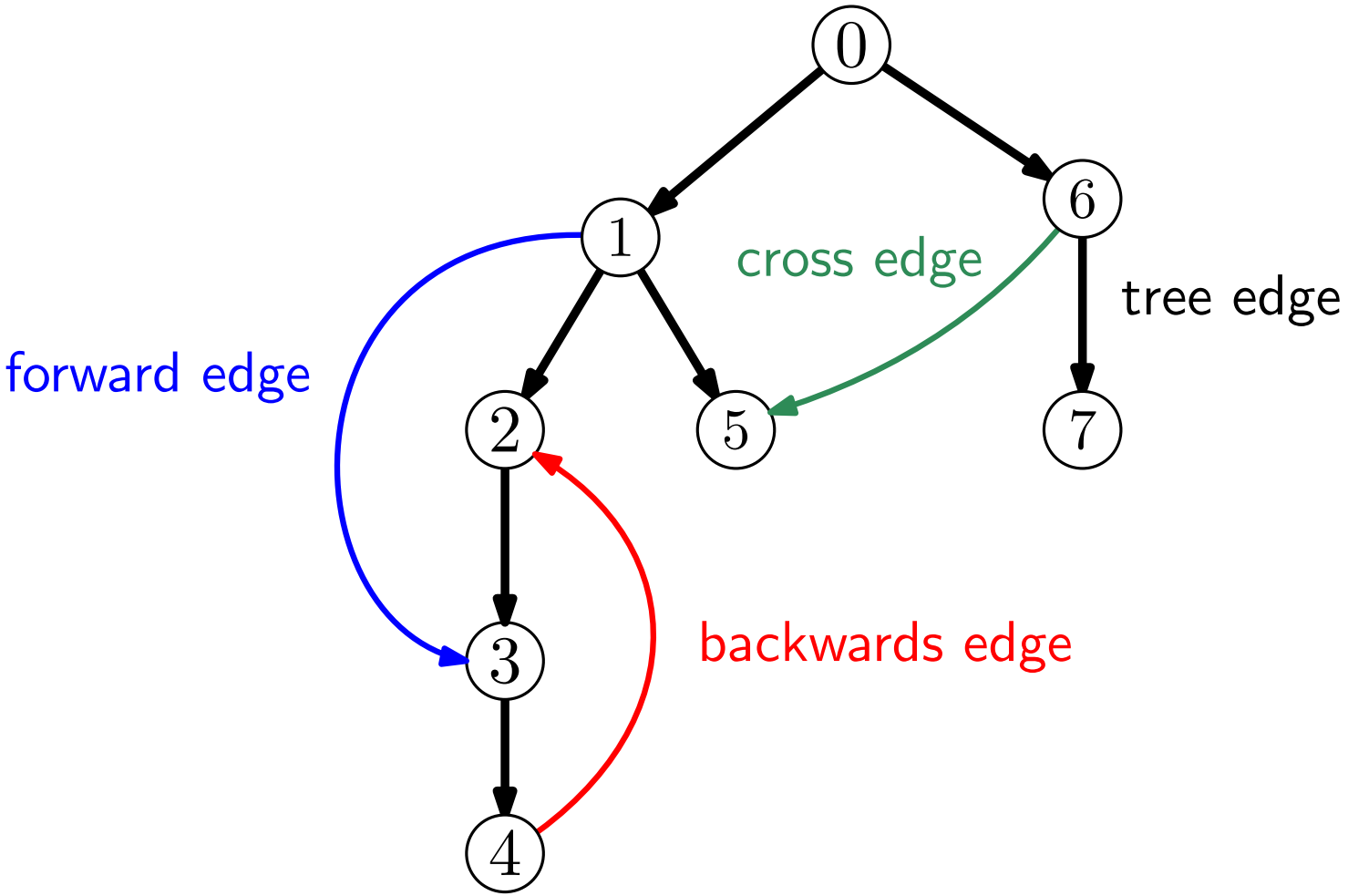
- `root_vertex` (input). The root vertex of the DFS. Default: the first vertex.
- `visitor` (input). Events are dispatched by invoking methods of a custom visitor object.

Some events:

- `discover_vertex(v, G)`: A new vertex `v` is discovered (visited).
- `examine_edge(e, G)`: An edge `e` is examined.
- `finish_vertex(v,G)`: All the neighbors of `v` have been examined.
- `tree_edge(e, G)`: An edge `e` of the DFS tree is discovered.
- `back_edge(e, G)`: A backwards edge `e` is discovered.
- `forward_or_cross_edge(e, G)`: A forward or cross edge `e` is discovered.

Time: $O(m+n)$

Depth First Search



A Custom DFS Visitor

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

class custom_visitor : public boost::default_dfs_visitor
{
public:
    void tree_edge(GT::edge_descriptor e, const Graph& G)
    {
        std::cout << "Discovered␣tree␣edge␣from␣"
                  << boost::source(e, G) << "␣to␣"
                  << boost::target(e, G) << "\n";
    }

    void back_edge(GT::edge_descriptor e, const Graph& G)
    {
        std::cout << "Discovered␣back␣edge␣from␣"
                  << boost::source(e, G) << "␣to␣"
                  << boost::target(e, G) << "\n";
    }
};
```

Starting the DFS Visit

```
Graph G; [...]

//Create an instance of our custom visitor
custom_visitor vis;

//Start the DFS visit
boost::depth_first_search(G, boost::root_vertex(0).visitor(vis));
```

Starting the DFS Visit

```
$ g++ -std=c++17 dfs.cpp -o dfs
$
$ ./dfs
Discovered tree edge from 0 to 2
Discovered tree edge from 2 to 1
Discovered back edge from 1 to 0
Discovered tree edge from 1 to 4
Discovered tree edge from 4 to 3
Discovered back edge from 3 to 2
$
```