# Библиотека для работы с графами

Разработаейте библиотеку для работы с графами. Библиотека должна быть описана в пространстве имен *au*. Эта библиотека должна включать:

1. непосредственно объект графа с произвольными дополнительными данными на ребрах и вершинах – заголовочный файл graph.h;
2. адаптер графа с фильтрацией по выбранному предикату вершин и ребер – заголовочный файл filtered_graph.h;
3. алгоритм поиска кратчайшего расстояним от одной вершины до другой вершины – заголовочный файл shortest_path.h.

## Требования к классу граф

1. Класс параметризуется данными вершин и данными ребер.
2. Граф не поддерживает мультиребер.
3. Граф ориентированный.
4. Класс поддерживает интерфейс хранения вершин, сходный с std::unordered_set:
    a. вершина не разделяется на ключ значение,
    b. существует реализация std::hash от типа вершины.
5. Типы данных вершин и ребер поддерживают копирование.
6. Класс графа конструируется по умолчанию, копируется, поддерживает семантику перемещения.
7. Реализует интерфейс вставки, удаления и перебора вершин и ребер графа посредством соответствующих итераторов.
8. Итераторы вершин и ребер возвращают данные соответственно вершин или ребер по ссылке соответствующей итератору константности (с исключением для итератора вершины, см. дальше).
9. vertex_iterator и vertex_const_iterator – одно и то же. Разыменовывание vertex_iterator’а дает константную ссылку.
10. Итератор по ребрам поддерживает методы from и to. Возвращают vertex_iterator.
11. Интерфейсы доступа (ровно как и итераторы) должны предоставляться как в константном, так и в неконстантном виде (но функций вида cbegin не требуется).
12. Будет достаточно, если у вас будут forward_iterator’ы.
13. Обратите, пожалуйста, внимание на то, что все итераторы имеют очень похожий интерфейс и реализацию, отличаются лишь несколькими стратегиями, такими как underline_iterator, операция разыменования и т.д. Требование: найдите, пожалуйста, способ обобщить реализации для итераторов, чтобы не писать несколько однотипных классов. Не используйте при этом макросы.
14. Константные итераторы могут конструироваться от неконстантных.
15. Ниже приводится минимальный интерфейс класса граф:

```
// 1. std::unordered_set-like interface for vertex, vertex is not
//    divided into key and value
// 2. vertex_data is hash'able by std::hash
// 3. not a multi graph
// 4. edge iterator supports 'from' and 'to' methods

template<class vertex_data, class edge_data>
struct graph
{
	// minimum interface

	// both const and non-const implementations
	using vertex_iterator = impl_defined;
	using edge_iterator   = impl_defined;

	using vertex_const_iterator = impl_defined;
	using edge_const_iterator   = impl_defined;

	// default constructable
	// also supports copy and move

	vertex_iterator add_vertex(
		/* allows passing vertex_data either
		by l-value reference or by value*/);

	edge_iterator   add_edge  (
		vertex_iterator const& from,
		vertex_iterator const& to,
		/* allows passing edge_data either
		by l-value reference or by value*/);

	void remove_vertex(vertex_iterator);
	void remove_edge  (edge_iterator  );

	// getters find, begin and end should support
	// both const and non-const versions
	vertex_iterator find_vertex(vertex_data const&);
	edge_iterator   find_edge  (vertex_iterator from, vertex_iterator to);

	vertex_iterator vertex_begin();
	vertex_iterator vertex_end  ();

	edge_iterator edge_begin(vertex_iterator from);
	edge_iterator edge_end  (vertex_iterator from);
};
```

## Требования к классу filtered_graph

Filtered graph представляет собой адаптер, позволяющий отсечь часть вершин или ребер из рассмотрения по какому-ту предикату. Такой адаптер параметризируется графом и предикатами

по вершинам и ребрам. Доступ только константный (константные и неконстантные итераторы суть одно и то же, но объявлены и те и те).

Ниже приводится минимальный интерфейс класса filtered_graph:

```
template<class graph, class vertex_filter, class edge_filter>
struct filtered_graph
{
	using vertex_data     = <vertex data from graph>
	using edge_data       = <edge data from graph>

	using vertex_iterator = impl_defined;
	using edge_iterator   = impl_defined;

	filtered_graph(
		graph const&, // do not make copy of graph
		/* filters either by l-value, or by reference */);

	// only const getters
	vertex_iterator find_vertex(typename graph::vertex_data const&) const;
	edge_iterator   find_edge  (vertex_iterator from, vertex_iterator to) const;

	vertex_iterator vertex_begin() const;
	vertex_iterator vertex_end  () const;

	edge_iterator edge_begin(vertex_iterator from) const;
	edge_iterator edge_end  (vertex_iterator from) const;
};
```

где vertex_filter и edge_filter – произвольные функторы, принимающие vertex_data и edge_data, возвращающие bool. True, если соответствующая data удовлетворяет требованию фильтра.

## Функция поиска кратчайшего расстояния

Реализуйте функцию поиска кратчайшего расстояния между двумя врешинами на графе. Функция принимает в качестве параметров:

1. сам граф,
2. вершины (от и до),
3. функтор вычисления длины от edge_data (для простоты, длина – вещественное число) и
4. функтор path_visitor, который последовательно вызовется для всех ребер пути, если путь будет найден, принимает edge_const_iterator.

Функция возвращает признак того, удалось ли найти путь. Ниже прототип функции:

```
// returns true, if path is found, false otherwise
template<class graph, class edge_len, class path_visitor>
bool find_shortest_path(
	graph const&,
	typename graph::vertex_const_iterator from,
	typename graph::vertex_const_iterator to,
	edge_len    && len_functor,
	path_visitor&& visitor);
```