

Задание №3 практикума по C++

В поддержку курса «Проектирование и программирование на C++»

Постановка задачи

Напишите библиотеку, с помощью которой можно будет представлять графы.

Основные типы графов, которые должны представляться в виде объектов:

- двудольный (Bipartite),
- полный (Complete),
- простой (Simple),
- взвешенный (Weighted).

Библиотека обязана поддерживать (путём переопределения операторов) некоторые псевдоарифметические выражения, которые собираются из таких объектов и арифметических операций над ними. При этом необходимо генерировать исключения `std::logic_error` в случае передачи в арифметические операции объектов с несовместимыми типами (см. [Перечень реализуемых арифметических операций](#)).

Обозначения вершин выполняется заглавными латинскими символами (A-Z). Соответственно, рёбра обозначаются парой заглавных латинских символов (AB, AD, и т.д.). При передаче на вход неправильных значений необходимо генерировать исключения `std::invalid_argument`.

Также у перечисленных типов графов должны поддерживаться методы:

- `ToString`, возвращающие строковое представление графа;
- `GetVertices`, возвращающие перечень вершин графа;
- `GetEdges`, возвращающее перечень рёбер графа.

Библиотека также обязана содержать функцию по нахождению кратчайшего пути между двумя заданными точками по *взвешенному* графу.

Ход выполнения задачи

1. Реализуйте иерархию классов основных функций, базовый класс `TGraph` из которой хранит [обязательные функторы](#) и фабрику по созданию объектов представления графов.

Фабрика объектов должна вызываться со строковым параметром `type`, который принимает значения из множества `{"bipartite", "complete", "simple", "weighted"}`, а также параметром, необходимым для определения графа выбранного типа, например: перечень рёбер, веса рёбер (см. [пример кода](#)).

2. В данной библиотеке реализуйте функцию, которая принимает на вход:

- взвешенный граф,
- вершину начала,
- вершину завершения,

...а далее находит с помощью одного из алгоритмов (на выбор; например, Дейкстры) кратчайший путь между двумя вершинами, и возвращает последовательный перечень ребер данного пути.

3. С использованием *gmock/gtest* напишите тесты для классов библиотеки, поддерживающих основные виды графов, и для операций составления псевдоарифметических выражений. Протестируйте создание объектов, генерацию ошибок, перечисление вершин и рёбер, формирование строкового представления. Напишите несколько тестов на нахождение кратчайшего пути (например, на связных и несвязных графах).

Примечание: рёбра **AB**, **BA** эквивалентны.

Примечание: если путь не может быть найден (например, граф не является связным), должна выполняться обработка данного случая.









Примечание: разрешается выполнение задания с презумпцией, что веса рёбер будут всегда неотрицательными.

Перечень реализуемых функторов

- `WeightedGraph TGraph.AsWeighted(int default_weight)`
 - представление графа в качестве взвешенного
 - аргумент `default_weight` - значение веса каждого ребра, для которого вес отсутствует;
- `const std::string TGraph.ToString()`
 - представление графа в качестве строки, содержащей тип графа и дополнительную информацию:
 - для *двудольного* графа: пара множеств вершин (*верхние*, *нижние*);
 - для *полного* графа: множество вершин;
 - для *простого* графа: множество рёбер;
 - для *взвешенного* графа: множество рёбер с их весами.
- `const std::vector<char> TGraph.GetVertices()`
 - получение множества вершин графа
- `const std::vector<std::vector<char>> TGraph.GetEdges()`
 - получение множества рёбер графа

Перечень реализуемых арифметических операций

- **сложение графов:**
 - **✓ двудольный (A) + двудольный (B) = двудольный (G):**
 - множество *верхних* (*нижних*) вершин **G** эквивалентно объединению множеств *верхних* (*нижних*) вершин **A** и **B**
 - **✓ полный (A) + полный (B) = полный (G):**
 - множество вершин **G** эквивалентно объединению множеств вершин **A** и **B**
 - **✓ взвешенный (A) + взвешенный (B) = взвешенный (G):**
 - множество рёбер **G** эквивалентно объединению множеств рёбер **A** и **B**
 - если ребро присутствует одновременно в **A** и в **B**, то в **G** сохраняется наименьший вес ребра
 - если ребро присутствует в **A** (**B**) и отсутствует в **B** (**A**), то ребро добавляется в **G** с сохранением веса ребра в **A** (**B**)

-  **взвешенный (A) + не взвешенный (B):**
 - логическая ошибка (веса рёбер не заданы для иных графов)
-  **не взвешенный (A) + взвешенный (B):**
 - логическая ошибка (веса рёбер не заданы для иных графов)
-  **не перечисленная пара типов графов (A, B) = простой (G):**
 - множество рёбер G эквивалентно объединению множеств рёбер A и B
- **вычитание графов:**
 -  **двудольный (A) - двудольный (B) = двудольный (G):**
 - множество *верхних* (*нижних*) вершин G эквивалентно разности множеств *верхних* (*нижних*) вершин A и B
 -  **полный (A) - полный (B) = полный (G):**
 - множество вершин G эквивалентно разности множеств вершин A и B
 -  **взвешенный (A) - любой (B) = взвешенный (G):**
 - множество рёбер G эквивалентно разности множеств рёбер A и B
 - веса рёбер, присутствующих в результирующем множестве рёбер G, переносятся из A
 -  **не взвешенный (A) - взвешенный (B) = не взвешенный (G):**
 - множество рёбер G эквивалентно разности множеств рёбер A и B
 -  **не перечисленная пара типов графов (A, B) = простой (G):**
 - множество рёбер G эквивалентно объединению множеств рёбер A и B

Пример кода

```

auto bipartite = graphFactory.Create("bipartite", {'A','B','C','D'}, {'E','F'});
std::cout << bipartite.ToString() << std::endl;
// BipartiteGraph {{A, B, C, D}, {E, F}}

auto complete = graphFactory.Create("complete", {'A', 'B', 'F'});
std::cout << complete.ToString() << std::endl;
// CompleteGraph {A, B, F}

auto simple = graphFactory.Create("simple", {'EF', 'FA'});
std::cout << simple.ToString() << std::endl;
// SimpleGraph {EF, FA}

auto weighted = graphFactory.Create("weighted", {'FD', 'ED'}, {5, 6});
std::cout << weighted.ToString() << std::endl;
// WeightedGraph {FD:5, ED:6}

auto p = *bipartite + *simple;
std::cout << w.ToString() << std::endl;
// SimpleGraph {AE, AF, BE, BF, CE, CF, DE, DF, EF, FA}

p -= *complete;
std::cout << w.ToString() << std::endl;
// SimpleGraph {AE, BE, CE, CF, DE, DF, EF, FA}

auto w = p.AsWeighted(1);
std::cout << w.ToString() << std::endl;

```

```
// WeightedGraph {AE:1, BE:1, CE:1, CF:1, DE:1, DF:1, EF:1, FA:1}

// пример: добавление рёбер простого к взвешенному
w += *simple;
// !!! std::logic_error !!!

// пример: удаление у взвешенного смежных с простым рёбер
w -= simple;
std::cout << w.ToString() << std::endl;
// WeightedGraph {AE:1, BE:1, CE:1, CF:1, DE:1, DF:1}

// пример: добавление/перезапись у взвешенного смежных с другим взвешенным рёбер
w += weighted;
std::cout << w.ToString() << std::endl;
// WeightedGraph {AE:1, BE:1, CE:1, CF:1, DE:6, DF:5}

// пример: удаление у взвешенного смежных с другим взвешенным рёбер
w -= weighted;
std::cout << w.ToString() << std::endl;
// WeightedGraph {AE:1, BE:1, CE:1, CF:1}
```