Класс *Graph* и алгоритм Краскала

Никитина Полина Владимировна
МФТИ ФПМИ Б05-924
Октябрь 2022

Содержание

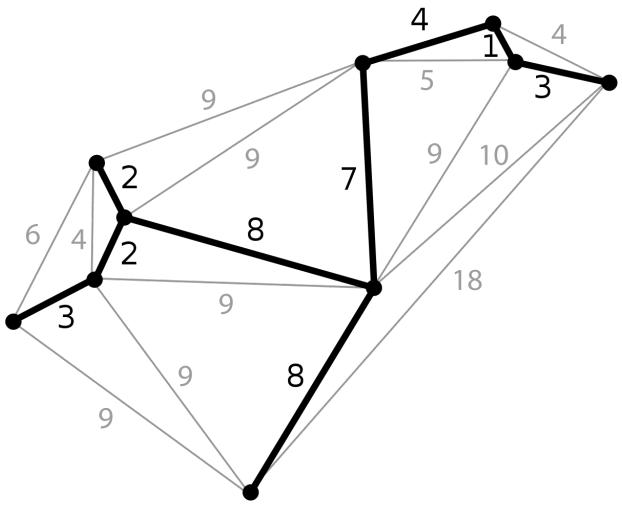
Теоретическая часть

Введение.

Графом является некоторая математическая абстракция реальной системы, объекты которой обладают парными связями.

Графы широко применяются в программировании и обработке изображений.

Одной из важнейших задач для графов является нахождение минимального остовного дерева в графе.



Постановка задачи.

Остовное дерево - это дерево, подграф данного графа, с тем же числом вершин, что и у исходного графа.

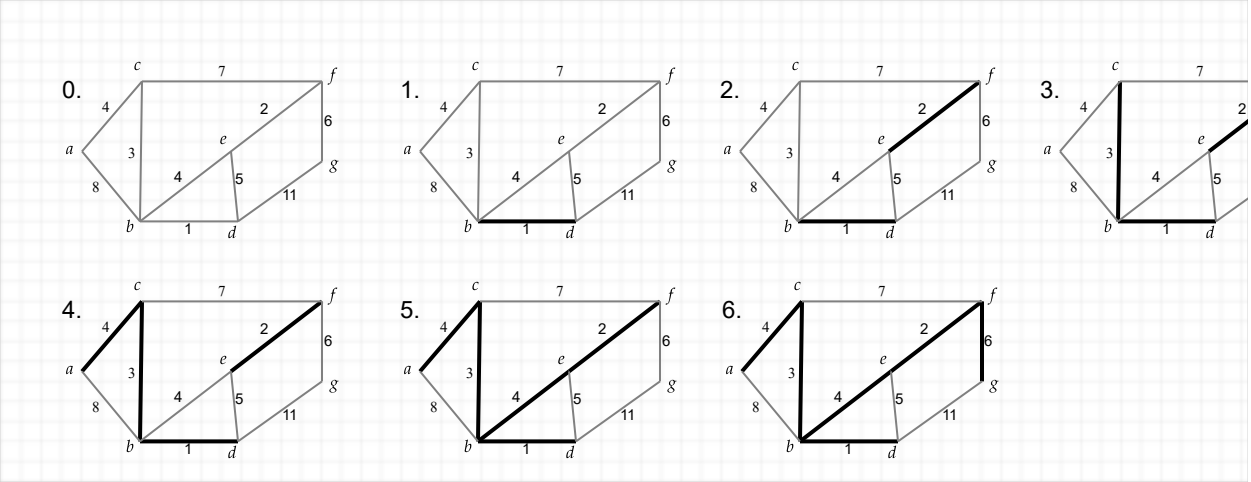
Минимальное остовное дерево в неориентированном взвешенном связном графе - это остовное дерево этого графа, имеющее минимальный возможный вес, где под весом дерева понимается сумма весов входящих в него рёбер.

Таким образом, у нас есть некоторый неориентированный граф $G = (V, E)$ с некоторой весовой функцией $w : E \to \mathbb{R}$ и мы хотим найти минимальное остовное дерево для G.

Описание алгоритма Краскала.

Алгоритм Краскала находит "безопасное" ребро для добавления в растущий лес путем поиска ребра (u, v) с минимальным весом среди всех ребер, соединяющих два дерева в лесу. Данный алгоритм является жадным, так как на каждом шаге он добавляет к лесу ребро с минимальным возможным весом.

Ниже привожу пример работы реализованного таким методом алгоритма.



В библиотеке реализован алгоритм Краскала с использованием непересекающихся множеств и эвристик для снижения времени работы функций непересекающихся множеств.

Непересекающиеся множества представляют собой корневые деревья без пересечений, то есть каждый член таких деревьев указывает только на родительский узел. Также у таких множеств есть несколько важных нам функций:

- создание множества из одного элемента;
- объединение двух множеств;
- поиск родительского узла для любого элеменета множества.

Эвристики, которые были использованы:

- объединение по рангу - подвешиваем меньшее дерево к большему
- сжатие пути - в процессе операции поиска родительского узла заставляет каждый узел дерева, через который проходит указывать на родительский узел.

```
    FindMST():
1   A = ∅
2   for v in this.vertex:
3       MakeSet(v)
4   sort(Graph.all_edges) by weight
5   for (u, v) in Graph.all_edges:
6       if (FindVertex(u) ≠ FindVertex(v)):
7           A = A ∪ (u, v)
8           Union(u, v)
9   return result
```

Вход: принимает на вход связный неориентированный взвешенный граф

Выход: минимальное остовное дерево графа

Ход алгоритма:

Для каждой вершины исходного графа создаем непересекающееся множество, состоящее из одной вершины.

Затем сортируем ребра графа по весу в неуменьшающемся порядке.

После этого для каждого ребра в отсортированном списке ребер проверяем относятся ли концы ребра к одному и тому же множеству или к разным множествам.

Когда находим ребро, концы которого относятся к разным множествам, то объединяем эти множества и начинаем искать следующее ребро до тех пор, пока в объединенном графе не будет столько же вершин, сколько было в исходном.

Последний граф, созданный объединением двух и является минимальным остовным деревом исходного графа.

Время работы: алгоритм Краскала в такой реализации работает за $O(E\ lgV)$.

Пользовательская документация

Библиотека и установка.

Ссылка на библиотеку: https://github.com/apollinaria-sleep/Graph

Чтобы использовать библиотеку необходимо:

1. Скачать себе репозиторий с кодом

```
git clone https://github.com/apollinaria-sleep/Graph
```

2. Собрать библиотеку:

```
mkdir build
cd build
cmake ./path // здесь необходимо указать путь к директории
make
```

Описание работы библиотеки.

Основным классом в библиотеке является класс Graph, который позволяет динамически работать с графами, то есть имеется возможность не только один раз инициализировать граф, но и добавлять или убирать вершины и ребра графа.

1. Инициализация
   - с помощью $std::vector\langle Edge\rangle$, где $Edge$ - класс ребер, который содержит в себе три поля: вершины ребра и его вес;

2. Добавление и удаление вершин
   - $void\ AddVertex(const\ int\&)$ добавление одной вершины по номеру без ребер
     Если вершина уже есть в графе, то выбрасывается исключение $std::exception$
   - $void\ AddVertex(const\ int\&,\ std::vector\langle int\rangle\&)$ добавление вершины по номеру и её ребер, идущих к вершинам которые уже есть в графе, с помощью списка $std::vector\langle int\rangle$ смежных вершин;
     Если вершина уже есть в графе, то выбрасывается исключение $std::exception$
   - $void\ AddVertex(const\ int\&,\ const\ std::vector\langle int\rangle\&, const\ std::vector\langle int\rangle\&)$ добавление вершины по номеру и её ребер, идущих к вершинам которые уже есть в графе, с помощью списка $std::vector\langle int\rangle$ смежных вершин с весами, которые также передаются с помощью списка;
     Если вершина уже есть в графе, то выбрасывается исключение $std::exception$
   - $void\ RemoveVertex(const\ int\&)$ удаление вершины по её номеру и всех её ребер;

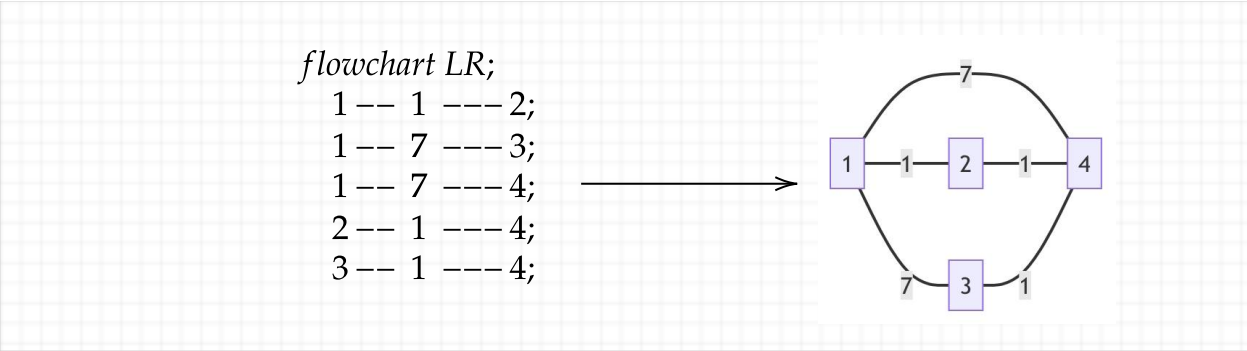
Если вершины нет в графе, то выбрасывается исключение *std* :: *exception*

3. Добавление и удаление ребер:
   - *void AddEdge*(*const Edge*&) добавление ребра, то есть объекта класса *Edge*;
     Если ребро уже есть в графе, то выбрасывается исключение *std* :: *exception*
   - *void AddEdge*(*const int*&, *const int*&) добавление в граф ребра с помощью указания двух его вершин;
     Если ребро уже есть в графе, то выбрасывается исключение *std* :: *exception*
   - *void AddEdge*(*const int*&, *const int*&, *const int*&) добавление в граф ребра с помощью указания двух его вершин и веса ребра;
     Если ребро уже есть в графе, то выбрасывается исключение *std* :: *exception*
   - *void RemoveEdge*(*const int*&, *const int*&) удаление ребра между некоторыми двумя вершинами;
     Если ребро уже есть в графе, то выбрасывается исключение *std* :: *exception*

4. Поиск минимального остовного дерева с помощью алгоритма Краскала:
   - *Graph FindMST*() находит минимальное остовное дерево в исходном графе и возвращает его копию;
     Если не удалось построить минимальное остовное дерево, то выбрасывается исключение *std* :: *exception*

5. Функции доступа к полям:
   - *int Size*() возвращает количество вершин в графе
   - *std* :: *vector*⟨*Edge*⟩ *AllEdges*() возвращает полный список ребер графа;
   - *std* :: *vector*⟨*int*⟩ *AllVertex*() возвращает полный список вершин графа;

6. Визуализация графа:
   - *void ShowGraph*(*std* :: *string*) - в указанный *.md* файл сохраняет представление графа
     Пример работы функции:



7. Функции ввода и вывода:

– $std :: istream\& \ operator\rangle\rangle(std :: istream\&, \ Graph\&)$ - формат ввода

```
vertex:
<кол-во вершин>
<номера вершин через пробел>
NotWeight/Weight
edge:
<кол-во ребер>
<описание ребер>
```

– $std :: ostream\& \ operator\langle\langle(std :: ostream\&, \ Graph\&)$ - формат вывода

```
vertex:
<кол-во вершин>
<номера вершин через пробел>
edge:
<описание ребер>
```

Тестирование.

Тесты покрывают все описанные выше функции библиотеки, кроме пунктов 6 и 7.

Чтобы запустить тесты из директории *build* запустите следующую команду:

```
./source/graph/graph_test
```

Техническая документация

Класс Graph

1.1 Введение

Класс Graph предназначен для работы с неориентированными, связными графами.
Ссылка на документацию

2.1 Class List

Здесь перечислены классы, структуры, объединения и интерфейсы с описанием класса:

3 Class Documentation

3.1 Edge Class Reference

Класс Edge реализует ребра графа.

#include ⟨graph.h⟩

Public Member Functions

- Edge (int from_v, int to_v)
- Edge (int from_v, int to_v, int weight)

Public Attributes

- int from_vertex
- int other_vertex
- int weight = 1

3.1.1 Detailed Description

Класс Edge реализует ребра графа.

Каждый объект класса Edge хранит в себе следующую информацию: from_vertex - одна из вершин ребра, other_vertex - другая вершина ребра, weight - вес ребра.

3.2 Graph Class Reference

Класс Graph основной класс, реализующий граф, поддерживающий добавление и удаление вершин и ребер, то есть способный динамически изменяться.

#include ⟨graph.h⟩

Public Member Functions

- Graph (const std::vector<Edge>& edge)
- Graph (const Graph& other)
- Graph& operator= (const Graph& other)
- Graph (Graph&& other) noexcept
- Graph& operator= (Graph&& other) noexcept
- void AddVertex(const int& v_num)
- void AddVertex(const int& v_num, std::vector<int>& edges)
- void AddVertex(const int& v_num, const std::vector<int>& edge, const std::vector<int>& weights)
- void RemoveVertex (const int& v_num)
- void AddEdge (const Edge& new_edge)
- void AddEdge (const int& from_v, const int& to_v)
- void AddEdge (const int& from_v, const int& to_v, const int& weight)
- void RemoveEdge (const int& from_v, const int& to_v)
- Graph FindMST ()
- int Size ()
- std::vector<Edge> AllEdges ()
- std::vector<int> AllVertex ()
- void ShowGraph (std::string file_name) const
- std::istream& ReadFrom (std::istream&)
- std::ostream& WriteTo (std::ostream&) const

3.2.1 Detailed Description

Класс Graph основной класс, реализующий граф, поддерживающий добавление и удаление вершин и ребер, то есть способный динамически изменяться.

Каждый объект класса Graph хранит в себе следующую информацию: vertex - std::vector<int> с вершинами графа, edge - std::vector<std::vector<Edge≫ матрица смежности графа, то есть множество ребер графа.

3.2.2 Constructor & Destructor Documentation

3.2.2.1 Graph()

```
Graph::Graph(const std::vector<Edge>& edge)
```

Создает объект класса Graph.

Parameters

edge	список ребер, которые задают граф

3.2.3 Member Function Documentation

3.2.3.1 AddEdge() [1/3]

```
void Graph::AddEdge (const Edge& new_edge)
```

Добавляет указанное ребро в граф.

Parameters

new_edge	ребро, которое нужно добавить в граф

Exceptions

std::exception	если ребро, которое хотим добавить, уже есть в графе

3.2.3.2 AddEdge() [2/3]

```
void Graph::AddEdge (const int& from_v, const int& to_v)
```

Добавляет ребро между двумя вершинами в граф.

Parameters

from_v	одна из вершин ребра
to_v	другая вершина ребра

Exceptions

std::exception	если ребро, которое хотим добавить, уже есть в графе

3.2.3.3 AddEdge() [3/3]

```
void Graph::AddEdge (const int& from_v, const int& to_v, const int&
weight)
```

Добавляет ребро с указанным весом между двумя вершинами в граф.

Parameters

from_v	одна из вершин ребра
to_v	другая вершина ребра
weight	вес ребра

Exceptions

std::exception	если ребро, которое хотим добавить, уже есть в графе

3.2.3.4 AddVertex() [1/3]

```
void Graph::AddVertex (const int& v_num)
```

Добавляет одну вершину в граф, обычно используется при добавлении первой вершины

Parameters

v_num	номер вершины

Exceptions

std::exception	если вершина, которую хотим добавить, уже есть в графе

3.2.3.5 AddVertex() [2/3]

```
void Graph::AddVertex (const int& v_num, std::vector<int>& edges)
```

Добавляет в граф вершину и её ребра. Необходимо, чтобы все вершины в списке уже находились в графе

Parameters

v_num	номер вершины
edges	список смежных вершин

Exceptions

std::exception	если вершина, которую хотим добавить, уже есть в графе

3.2.3.6 AddVertex() [3/3]

```
void Graph::AddVertex (const int& v_num, const std::vector<int>& edges,
const std::vector<int>& weights)
```

Добавляет в граф вершину и её ребра с весами. Необходимо, чтобы все вершины в списке уже находились в графе

Parameters

v_num	номер вершины
edges	список смежных вершин
weights	список весов ребер

Exceptions

std::exception	если вершина, которую хотим добавить, уже есть в графе

3.2.3.7 AllEdges()

```
std::vector<Edge> Graph::AllEdges()
```

Функция, показывающая ребра графа

Returns

Список всех ребер графа

3.2.3.8 AllVertex()

```
std::vector<int> Graph::AllVertex()
```

Функция, показывающая вершины графа

Returns

Список всех вершин графа

3.2.3.9 FindMST()

```
Graph Graph::FindMST ()
```

Функция поиска минимального остовного дерева в связном, взвешенном графе

Returns

Объект класса Graph, являющийся минимальным остовным деревом исходного графа

Exceptions

std::exception	если на вход был подан некорректный граф, для которого нельзя построить минимальное остовное дерево

3.2.3.10 RemoveEdge()

```
void Graph::RemoveEdge (const int& from_v, const int& to_v)
```

Удаляет ребро между двумя вершинами

Parameters

from_v	одна из вершин ребра
to_v	другая вершина ребра

Exceptions

std::exception	если ребра, которое хотим удалить, нет в графе

3.2.3.11 RemoveVertex()

```
void Graph::RemoveVertex (const int& v_num)
```

Удаляет вершину из графа вместе со всеми её ребрами

Parameters

v_num	номер вершины

Exceptions

std::exception	если в графе нет вершины, которую хотим удалить

3.2.3.12 ShowGraph()

```
void Graph::ShowGraph (std::string file_name) const
```

Визуализирует граф

Parameters

file_name	имя .md файла, в который будет записано описание графа

Note

После компиляции .md файла будет получено визульное представление графа

3.2.3.13 Size()

```
int Graph::Size()
```

Функция определения размера графа

Returns

Количество вершин в графе

3.3 Set Class Reference

Вспомогательный класс Set непересекающихся множеств для эффективной реализации алгоритма Краскала.

Public Member Functions

- void MakeSet (const size_t& ind)
- void Union (Set* other)
- Set* FindSet ()

3.3.1 Detailed Description

Вспомогательный класс Set непересекающихся множеств для эффективной реализации алгоритма Краскала.
Каждый объект класса Set хранит в себе следующую информацию: root - указатель на Set, который является корневым для поддерева, rank - высота поддерева

3.3.2 Member Function Documentation

3.3.2.1 FindSet()

```
Set* Set::FindSet () [inline]
```

Определяет к какому множеству относится поддерево
Returns

Указатель на объект класса Set, который является корневым для поддерева

3.3.2.2 MakeSet()

```
void Set::MakeSet(const size_t& ind) [inline]
```

Создает объект класса Set для одной вершины
Parameters

ind	номер вершины

3.3.2.3 Union()

```
void Set::Union(Set* other) [inline]
```

Объединяет одно множество с другим
Parameters

other	указатель на другой объект класса Set

Используемая литература

Кормен Т. Алгоритмы. Построение и анализ / Кормен Т., Лейзерсон Ч. - 3-е изд. - М.:Диалектика-Вильямс, 2019. - 1328 с.