Класс *Graph* и алгоритм Краскала

Никитина Полина Владимировна
МФТИ ФПМИ Б05-924
Октябрь 2022

Содержание

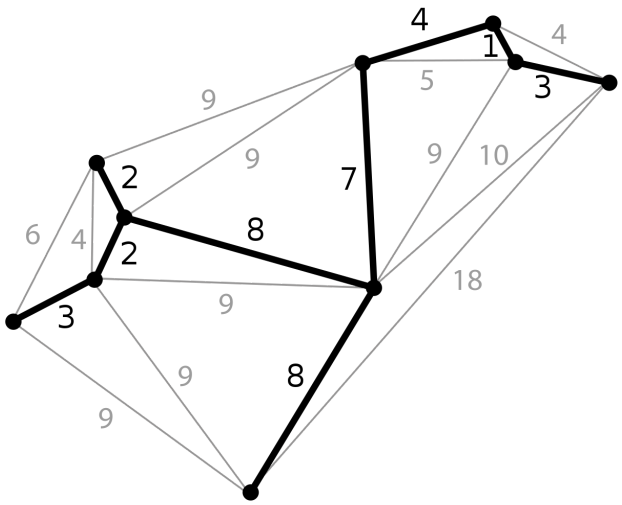
Теоретическая часть

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.

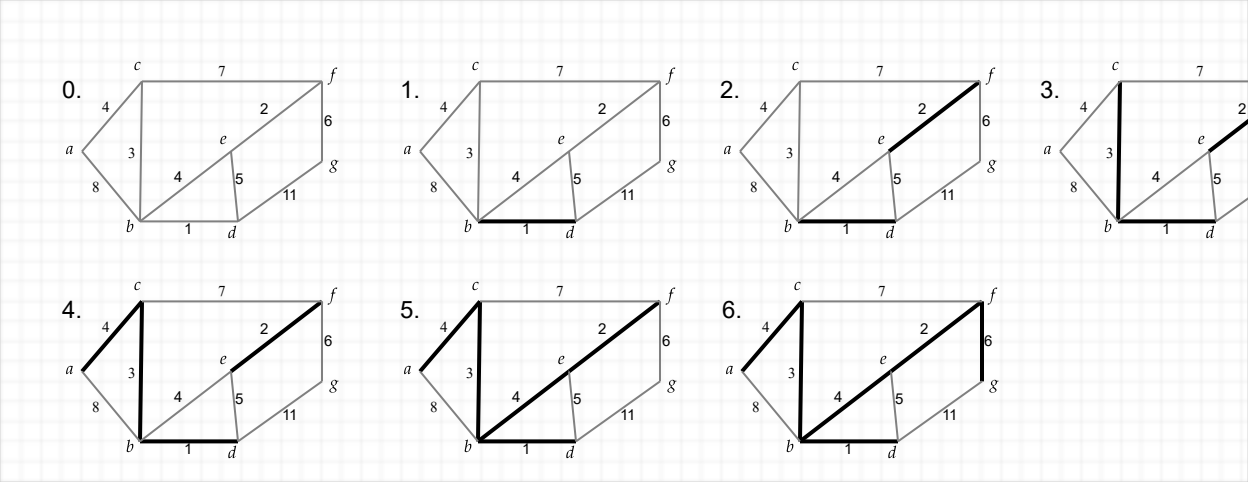
Минимальное остовное дерево в неориентированном взвешенном связном графе - это остовное дерево этого графа, имеющее минимальный возможный вес, где под весом дерева понимается сумма весов входящих в него рёбер.

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;

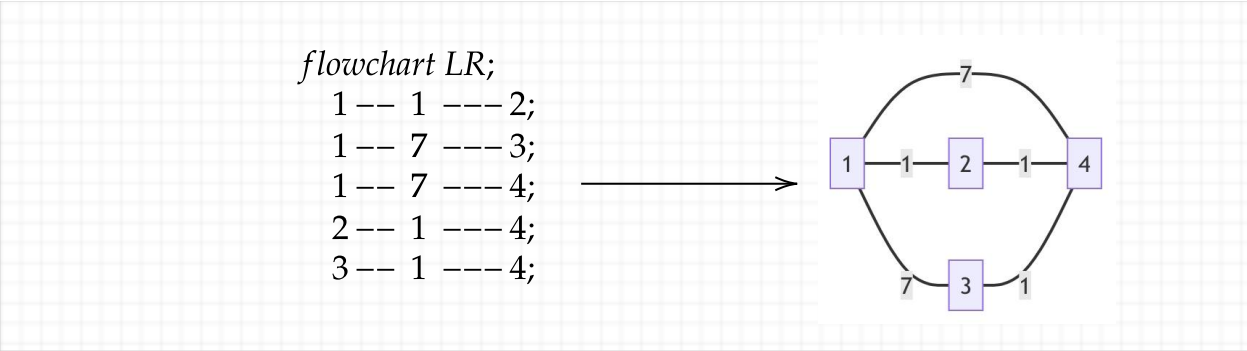
Если вершины нет в графе, то выбрасывается исключение *std* :: *exception*

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

Техническая документация находится в папке html файл index.html

Используемая литература

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