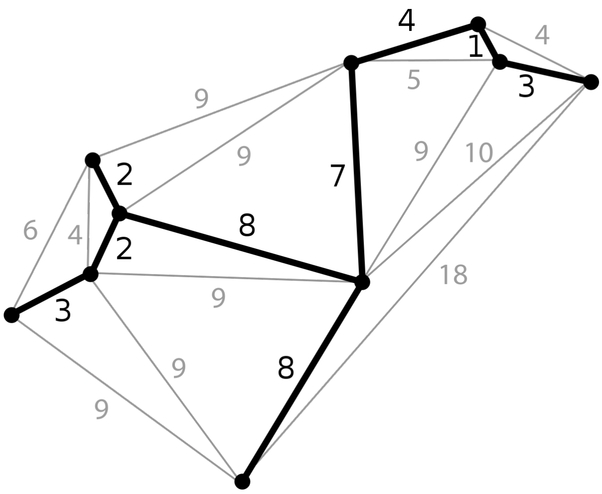
Построение минимального остовного дерева



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.

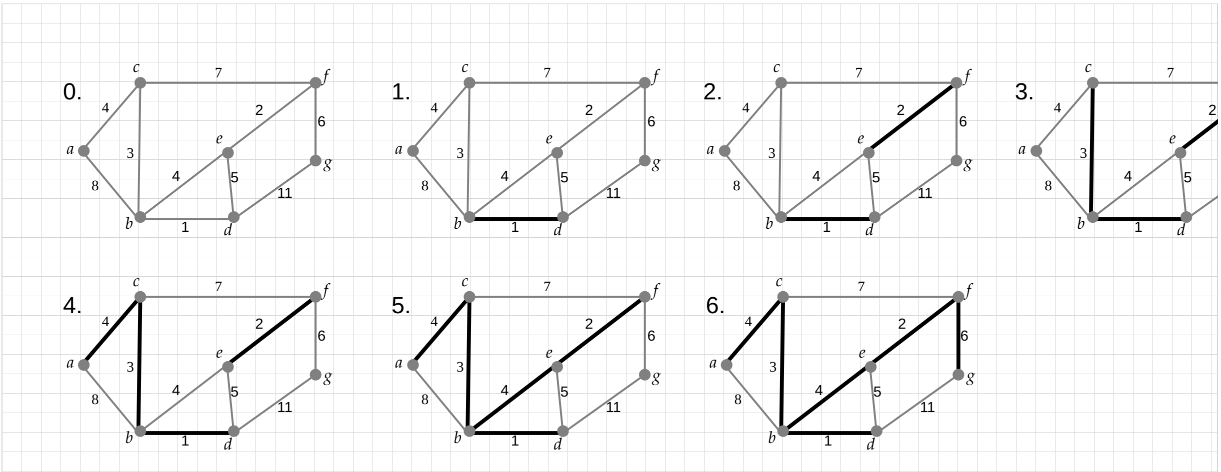
Таким образом, у нас есть некоторый неориентированный граф $G = (V, E)$ с некоторой весовой функцией $w : E \to \mathbb{R}$ и мы хотим найти минимальное остовное дерево для G.

Важно отметить, что представленный мною алгоритм применим только для связного, неориентированного графа, но условие взвешенности не обязательно. В этом случае, алгоритм будет при вызове функции $FindMST()$ возвращать некоторый подграф исходного графа, содержащий все его вершины.

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.



Моя реализация алгоритма Краскала использует структуру непересекающихся множеств, при этом эти множества являются в то же время подграфами исходного графа (деревьями в лесу).
Чтобы осуществить такую реализацию я создала собственный класс $Graph$, о котором расскажу в следующем разделе.

Класс $Graph$

Имеется два стандартных способа представления графа: как набора списков смежных вершин и как матрицы смежности. Я реализовала свой класс с помощью второго варианта, так как он обычно предпочтительнее, так как обеспечивает компактное представление разреженных графов.

Мой класс $Graph$ содержит в себе набор списков ребер (то есть некоторый объектов класса $Edge$, это потребовалось, так как алгоритм работает со взвешенными графами), он реализован с помощью $std :: vector$.
Кроме того, я добавила в свой класс отдельный список вершин. Это сделано по нескольким причинам:

- в реализации алгоритма Краскала мы хотим работать со структурой множеств, список вершин и помогает мне работать с ней;
- также я хотела, чтобы класс $Graph$ поддерживал операции добавления и удаления вершин и ребер, а вершины были не просто проиндексированы, а имели свои собственные номера, возможно, не связанные с индексацией;

В итоге объект класса $Graph$ имеет следующие методы:

- $AddVertex$ - добавляет в граф вершину или без ребер, или с ними;
- $AddEdge$ - добавляет в граф ребро между двумя вершинами;
- $RemoveVertex$ - удаляет вершину и все её ребра;
- $RemoveEdge$ - удаляет ребро графа;
- $VertexNeighbours$ - возвращает список соседей вершины;
- $Size$ - возвращает количество вершин в графе;
- $Union$ - объединяет два графа в один, в основном применяется в алгоритме Краскала;
- $FindMST$ - функция поиска минимального остовного дерева с помощью алгоритма Краскала.

Также класс поддерживает ввод и вывод в поток.

Алгоритм Краскала

Вход: принимает на вход корректный объект класса $Graph$ (то есть связный неориентированный граф)

Выход: возвращает объект класса $Graph$, который будет являться минимальным остовным деревом для исходного класса.

Ход алгоритма:

Для каждой вершины исходного графа создаем объект класса $Graph$, состоящий из одной вершины - это те же самые множества, так как каждый объект класса $Graph$ хранит в себе список вершин и такие объекты можно объединять, кроме того в классе $Graph$ есть $private -$ функция $findVertex$, которая позволяет методам определить содержится ли вершина в множестве.

Затем объединяем все ребра исходного графа в массив и сортируем его по весу ребер в неуменьшающемся порядке (это я выполнила с помощью $std::sort$).

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.

Приведу псевдокод здесь, чтобы проще было далее считать асимптотику получившегося алгоритма:

```
   FindMST() :
1  for v in this.vertex :
2      Graph g; g.AddVertex(v)
3  sort(Graph.all_edges) by weight
4  for (u, v) in Graph.all_edges :
5      if (Graph.findVertex(u) ≠ Graph.findVertex(v)) :
6          result = g1.Union(g2)
7  return result
```

Оценка сложности

В оригинальном алгоритме Краскала используются специальные множества, чтобы эффективно объединять по рангу и сжимать путь, в своей реализации я использовала обычные массивы, поэтому в этом случае асимптотика будет значительно отличаться.

Время необходимое для сортировки массива ребер в строке 3 занимает $O(E \, log \, E)$.

Цикл for в строках 5,6 выполняет $O(E)$ операций $findVertex$ и $Union$ над лесом графов. Одна операция $findVertex$ занимает $O(V)$ времени, так как реализована она простым проходом по списку вершин. Одна операция $Union$ занимает $O(VE)$ времени, так как я сохраняю не только вершины, но и их ребра, чтобы поддерживать структуру графа.

Операция создания первоначального графа для каждой вершины занимает $O(1)$.

Получаем, что время работы моего алгоритма можно оценить как $O\left(E^2 V\right)$.

Получили довольно плохую асимптотику, но смогли на всех этапах алгоритма поддерживать структуру графа.

Используемая литература

Идея алгоритма и базовые принципы создания класса была почерпнуты в книге
Т.Кормен "Алгоритмы. Построение и анализ", третье издание
(Глава 22. Элементарные алгоритмы для работы с графами, стр.626-629; Глава 23. Минимальные остовные деревья, стр. 661-680)