

Graphs

Interlude

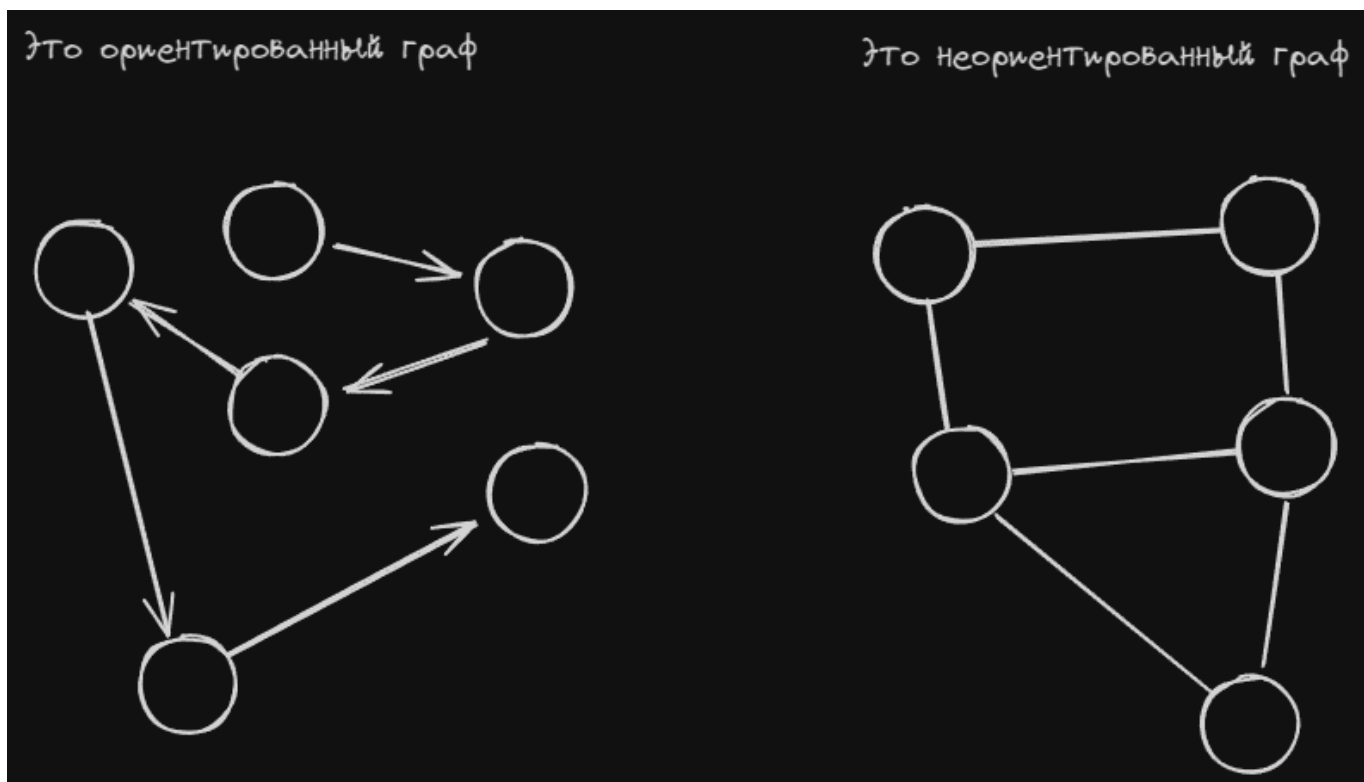
С ростом количества данных появилось желание удобно работать с ними, различать их по типам (классифицировать) и визуализировать. Представьте, что вам необходимо хранить информацию о связях в социальной сети. Было бы крайне неэффективно хранить структуру данных в виде

```
{
  "users": [
    {
      "userId": 1,
      "friendId": 2
    },
    {
      "userId": 1,
      "friendId": 3
    },
    {
      "userId": 1,
      "friendId": 4
    },
    ...
  ]
}
```

Графы это инструмент, который позволяет переносить математические модели в программу, интерпретация поставленной задачи. Например тоже самое справедливо для очереди, деревьев итп.

So what

Графы — очень простая вещь, это просто множество вершин и ребер. Ну если удобно, то это стрелочки (линии) и точки. Максимально простая вещь. Графы бывают ориентированные и неориентированные — со стрелочками и с линиями соответственно.



Возвращаясь к проблеме с социальной сетью, можно применить графы для хранения таких данных, это было бы компактнее и удобнее. И в данном случае возникает закономерный вопрос, какой тип графов использовать..

Graph types 🌀

В теории графов выделяют несколько категорий графов

- Ориентированные
- Неориентированные
- Мультиграфы
- Полные
- Псевдографы
- Взвешенные
- и другие возможные комбинации выше перечисленного

Из всего этого чаще всего придется работать с *ориентированными*, *неориентированными* и *взвешенными*. Единственное, что осталось неосвещенным — взвешенные графы. Здесь тоже все просто — это графы, которые имеют "весовую" характеристику.

В случае с социальной сетью, лучше всего будет использовать неориентированный граф, потому что он чаще всего определяет симметричное отношение на множестве вершин. Очевидно, что если люди не корыстные, то дружба в реальной жизни двусторонняя, а в социальной сети другой возможности оказаться во френдлисте не бывает.

А если необходимо вместе с этим хранить список подписчиков пользователя? Ну а тут уже неплохо бы хранить ориентированный граф, поскольку они используются в моделях, где хочется хранить состояние.

Complexity 📈

Раньше все алгоритмы оценивались с помощью одной функциональной переменной и на то была причина — сложность зависела от одного набора входных данных.

В графах хранится много данных и основными характеристиками любого графа являются вершины (Vertices) и ребра (Edges). Поэтому в алгоритмах на графах, как правило, используют две переменные в ходе оценки асимптотической сложности. Отсюда выходит очень много забавных свойств, которые применяются как в алгоритмах, так и в теории графов.

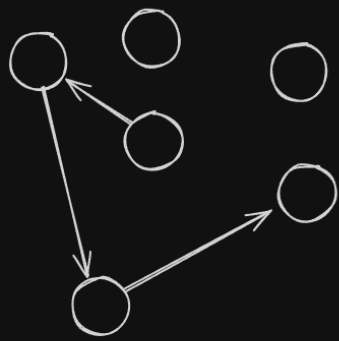
Для начала достаточно будет понимать, как связаны между собой вершины и ребра..

Graph coupling

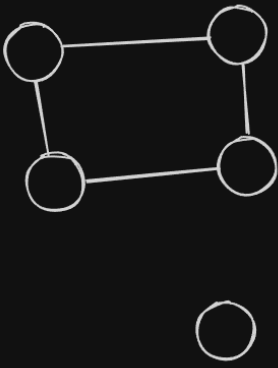
Связи вообще зависят от типа графа, но тут тоже все просто. Прежде чем понять, в каком отношении находятся вершины и ребра в графе, нужно определить термин **связности графа**.

Связность графа — это факт существования пути от любой вершины к любой другой. Ну а если менее формально, то это простое человеческое желание, чтобы граф не распался на куски.

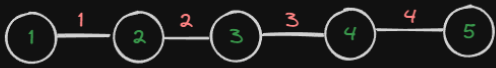
Это несвязный ориентированный граф



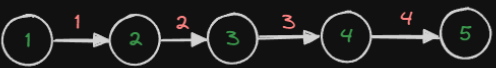
Это несвязный неориентированный граф



Для неориентированных графов



Для ориентированных графов



Если хочется, чтобы граф был связным, нужно потребовать выполнение

$E \geq V - 1$

При этом граф будет без петель и параллельных ребер, если

$E \leq \frac{V * (V - 1)}{2}$

Если хочется, чтобы граф был связным, нужно потребовать выполнение

$E \geq V - 1$

При этом граф будет без петель и параллельных ребер, если

$E \leq V * (V - 1)$