Ациклические графы

01/04/2023

В прошлой серии

- Граф - это множество вершин и ребер между ними
- Связность вершин определяется наличием пути по ребрам из одной в другую
- Компоненты связности можно обходить (в ширину или в глубину)
- В графах можно считать расстояние как сумму весов на пути, в которой всплывает динамика

$$dist_v = \min_u(dist(u) + w(v, u))$$

- Дейкстра, BFS, Форд-Беллман по-разному считают эту динамику исходя из конкретных ограничений в задаче

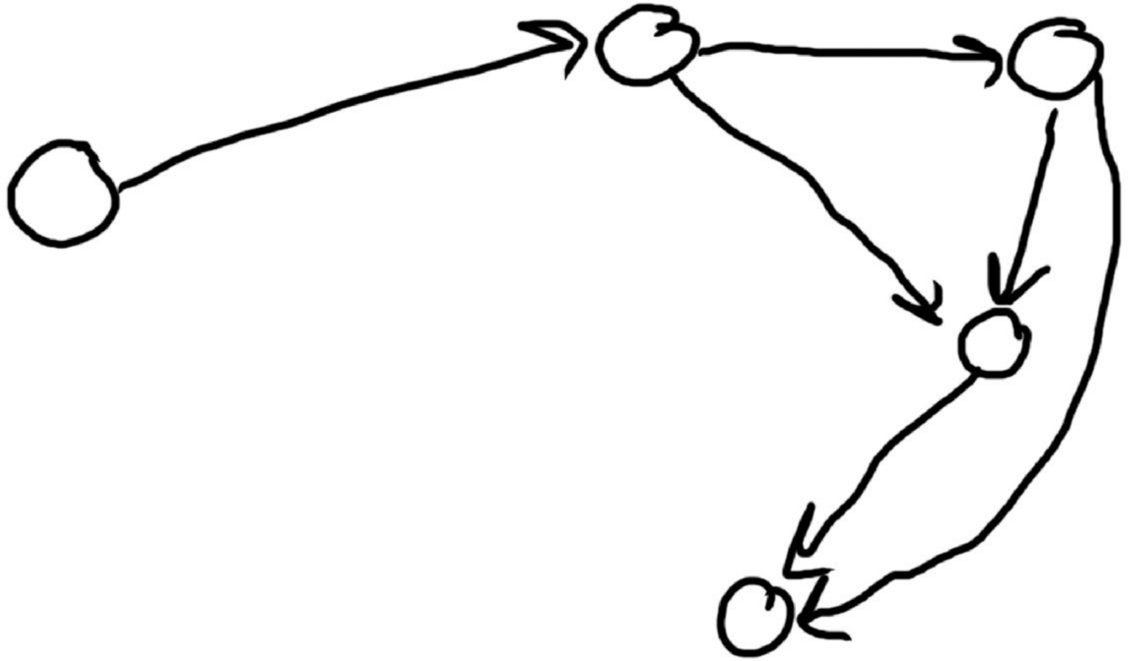
Неориентированный случай

В неориентированном случае ациклический граф это дерево. Это большая и интересная тема для изучения, но не для этой лекции.

Keywords: Дерево эйлерова обхода, LCA, HLD, CD, Link-Cut Tree

DAG

Directed Acyclic Graph - ориентированный граф без циклов



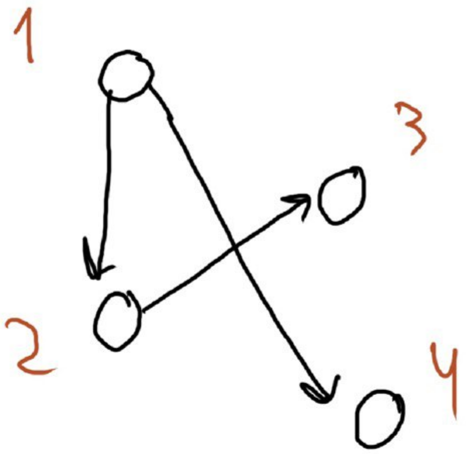
Топологическая сортировка

Пусть вершина v соединена ребрами с $u_1, \ldots, u_k$. Тогда из каждой вершины u_i нет пути в v.

Определим *топологическую сортировку* вершин как такой порядок, при котором вершина v всегда раньше всех своих соседей u_i.

Вопрос: что мы знаем про последнюю вершину в порядке топологической сортировки?

Разные варианты



1 4 2 3

1 2 4 3

1 2 3 4

Построение топологической сортировки с помощью dfs

Строим ответ рекурсивно: сначала последовательность для детей, а потом для себя.

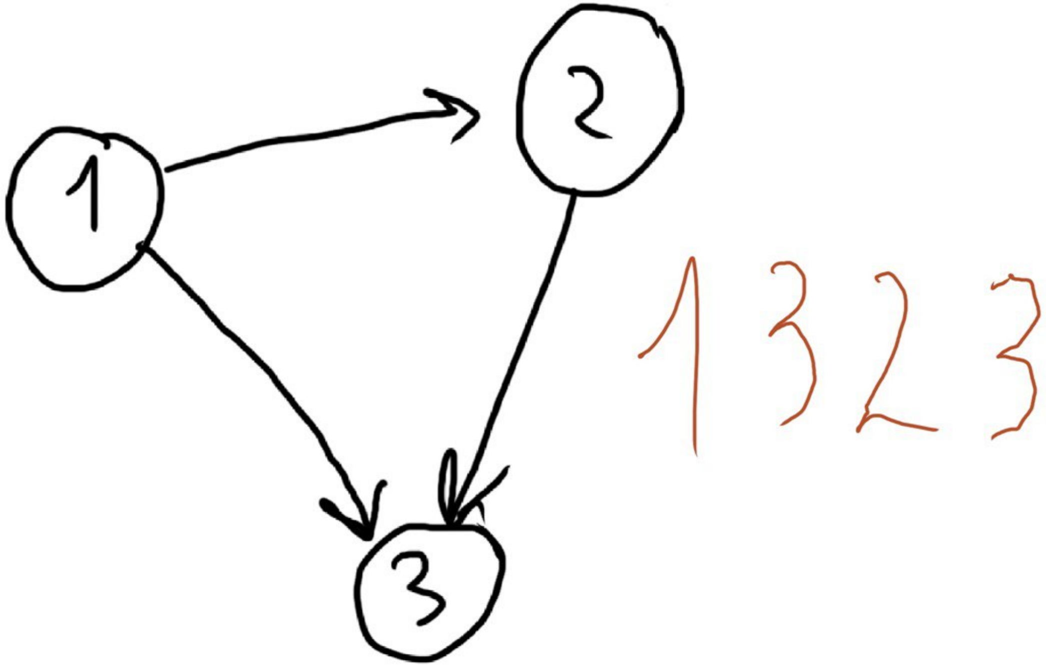
Хотим примерно такой алгоритм, который нам будет явно гарантировать то, что в сортировке дети встретятся позже родителей

```
topsort = []
def dfs(v):
    for u in g[v]:
        dfs(u)
    topsort.append(v)

dfs(0)
topsort = topsort[::-1]
```

Вопрос: Какая проблема с этим алгоритмом?

Дубликаты



Убираем дубликаты

```
topsort = []
used = [False] * n
def dfs(v):
    used[v] = True
    for u in g[v]:
        if !used[u]:
            dfs(u)
    topsort.append(v)

dfs(0)
topsort = topsort[::-1]
```

Корректность построения

Мы знаем, что для тех ребер, по которым делал переход dfs, порядок явно сохранен.

Если мы отсекли переход, то вершину u мы проходили раньше, а значит она в итоговом списке будет позже v.

Поскольку свойство выполняется для всех ребер, то оно выполняется и для всех путей: все пути идут слева направо.

Приплетаем ДП

Вспомним, из чего состоит динамика

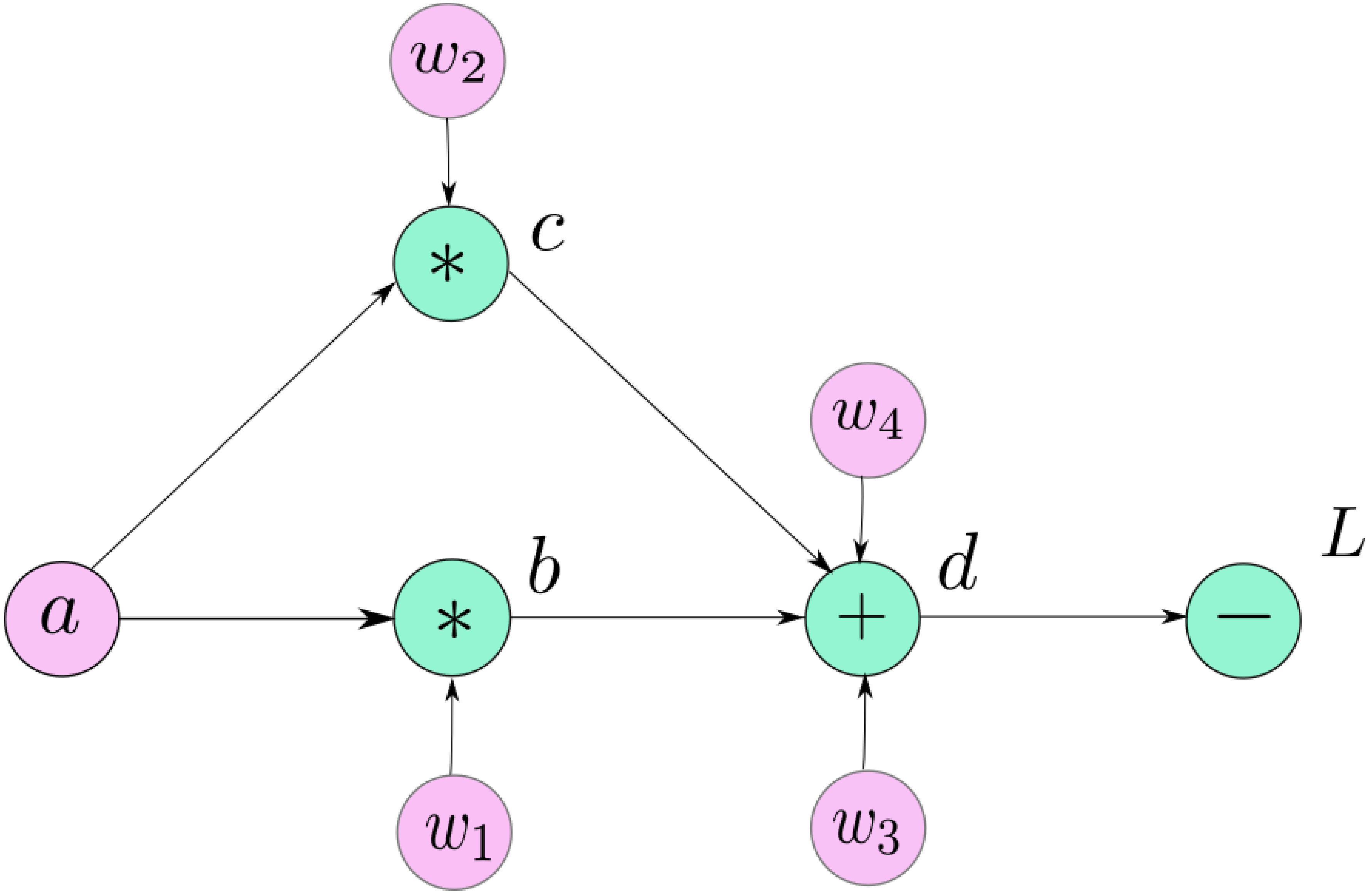
1. **Состояние**
2. **Переходы**
3. **Порядок пересчета**
4. База
5. Результат

Пункты 1-3 это DAG вычислений в явном виде!

DP = DAG вычислений

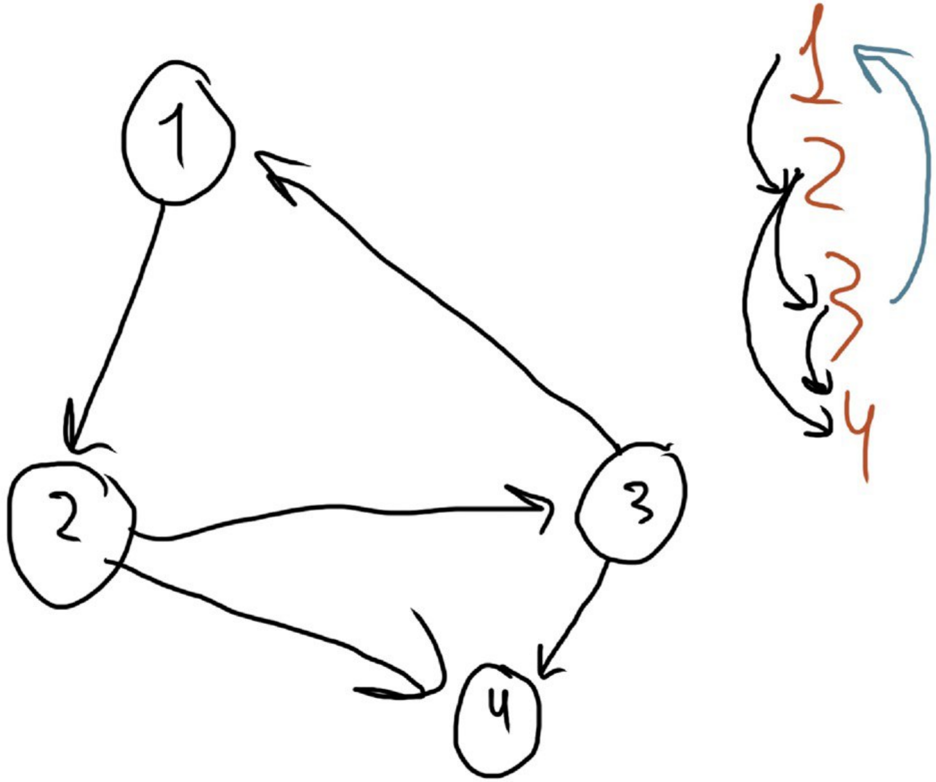
Состояние динамики соответствует вершинам, а переходы соответствуют ребрам. Порядок пересчета существует, потому что граф ациклический, и его можно задать топологической сортировкой.

Байка: PyTorch



Обратные ребра

Алгоритм поиска топологической сортировки можно запустить и на обычном графе, но тогда у нас нарушится свойство, что ребра есть только ~~слева направо~~ сверху вниз



Компоненты сильной связности

Идея: мы хотим ввести порядок на графе, даже если порядок в явном виде задать нельзя из-за циклов.

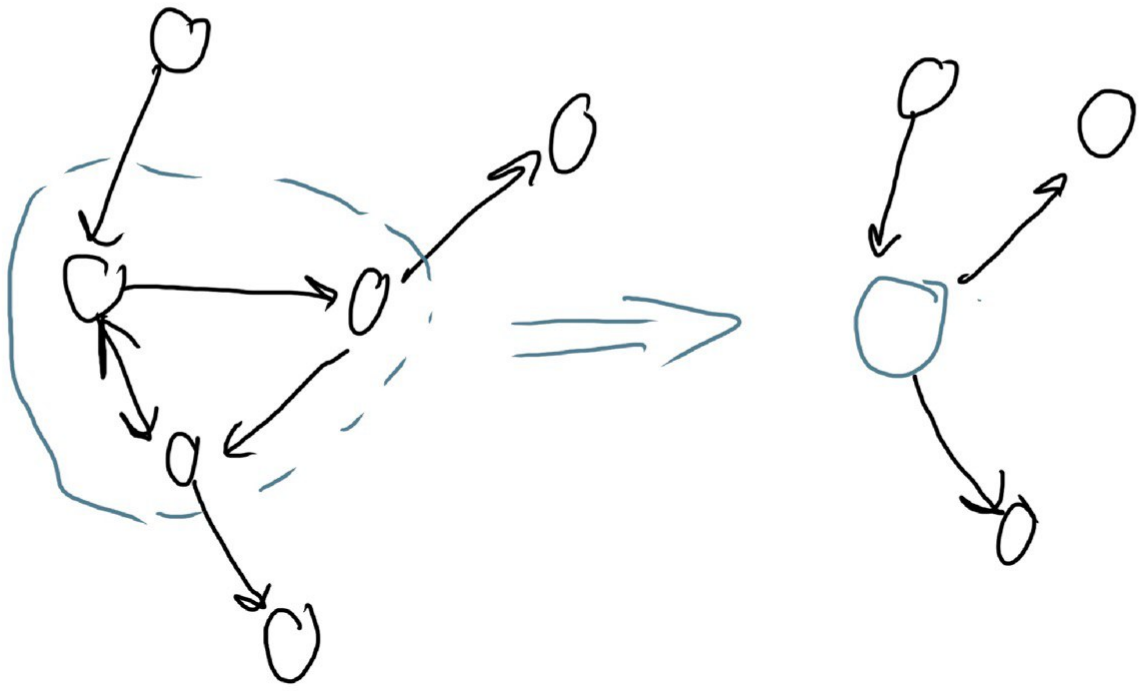
Давайте сожмем циклы в компоненты сильной связности. Иначе говоря, мы будем считать, что все вершины в цикле неотделимы.

Интуиция такая: мы можем считать, что у вершин в одной компоненте сильной связности одинаковые характеристики

Цель

- Найти компоненты связности
- Сжать их в одну вершину
- Составить *индуцированный* DAG

Компонента сильной связности



Алгоритм Kosaraju (конденсация графа)

Наблюдение: обратное ребро в топологической сортировке сжимает какой-то цикл. Если справа налево есть ребро $u \to v$, то из v достижимо u (из топсорта), и из u достижимо v.

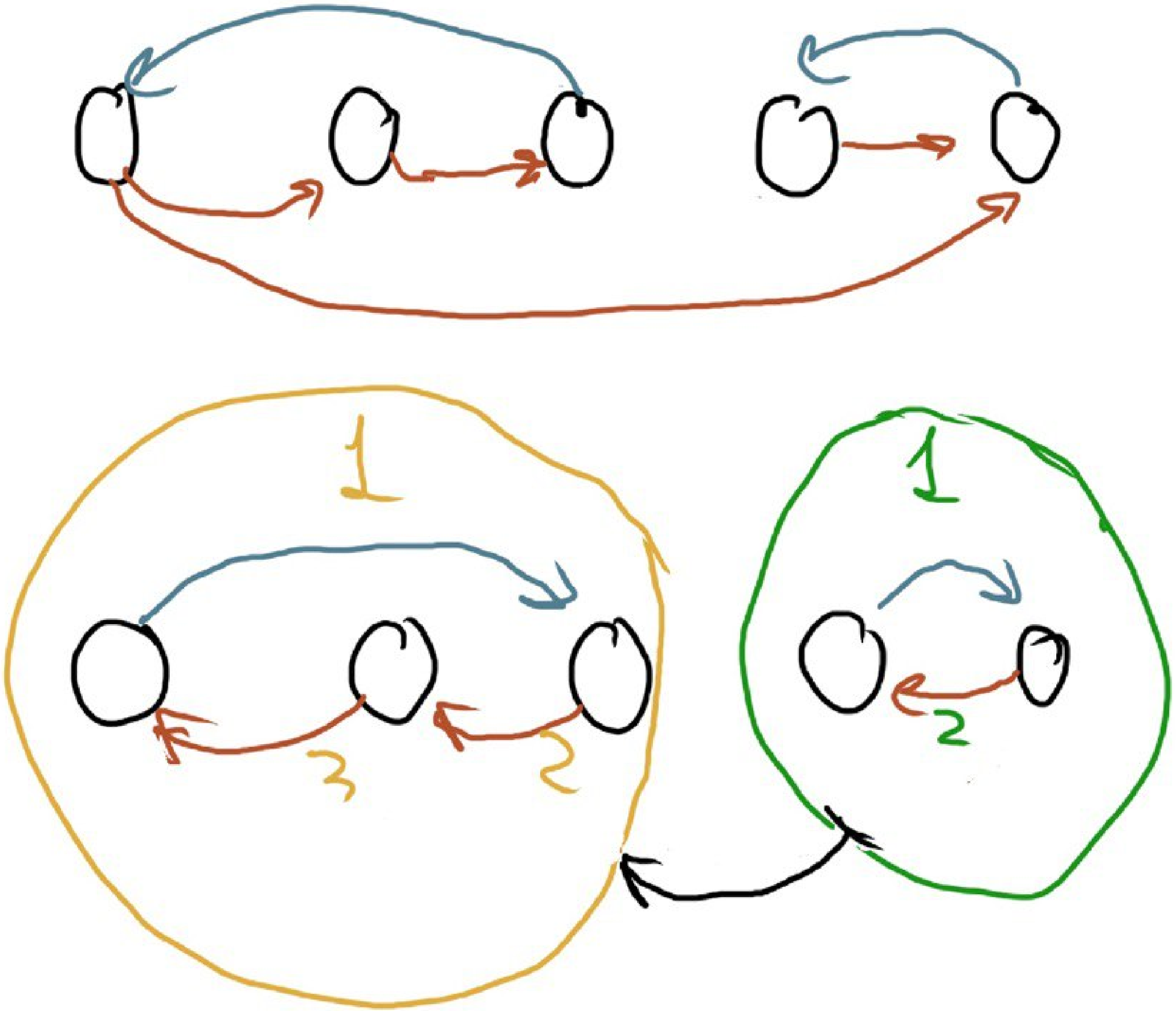
1. Сделаем топологическую сортировку графа
2. Развернем все ребра графа (получив `gr[v]`)
3. Выделим для каждой непомеченной v в компоненту все вершины справа от v, связанные с v в `gr`.

Реализация

```
# g, gr = ...
# calculate topsort

color = []
g_res = [{} for _ in range(n)]
def unite(v, c):
    color[v] = c
    for u in gr[v]:
        if color[u] == 0:
            unite(u, c)
        elif color[u] != c:
            g_res[color[u]].insert(c)

c = 1
for v in topsort:
    if color[v] == 0:
        unite(v, c)
        c += 1
```


1
3
2
1
2

2-SAT

Задача: найти такие $x_i \in [0, 1]$, что выполяется КНФ, в каждой скобке которого по 2 переменные

Например,

$$(x_0 \vee \overline{x}_1) \wedge (x_1 \vee x_2)$$

выполняется при $x = (1, 1, 1)$

Определяем вершины

Сделаем $2n$ вершин - для каждой x_i заведем вершину со "значением" 1 и 0.

Будем считать, что мы хотим "выбрать" из этого множества n вершин, по одной из каждой пары.

Строим ребра

Добавим ограничения:

Если есть $(x_i \vee x_j)$, это ограничение значит что если $x_i = 0$, то x_j обязательно 1 и наоборот.

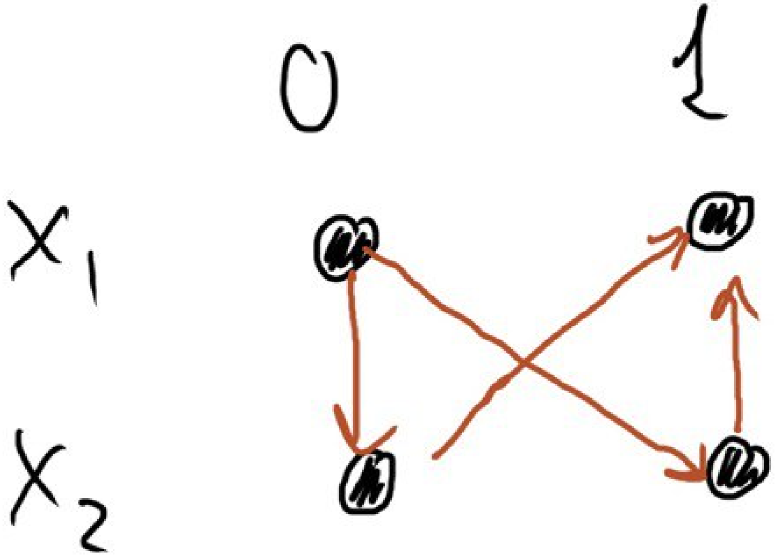
Таким образом, мы создаем два ребра:

$$\overline{x}_i \to x_j, \overline{x}_j \to x_i$$

Тогда, если мы зафиксируем значение какой-то переменной, достижимые из нее вершины тоже фиксируются.

Пример

$$(x_1 \vee x_2) \wedge (x_1 \vee \overline{x}_2)$$



$$(x_1 \vee x_2) \wedge (x_1 \vee \bar{x}_2)$$

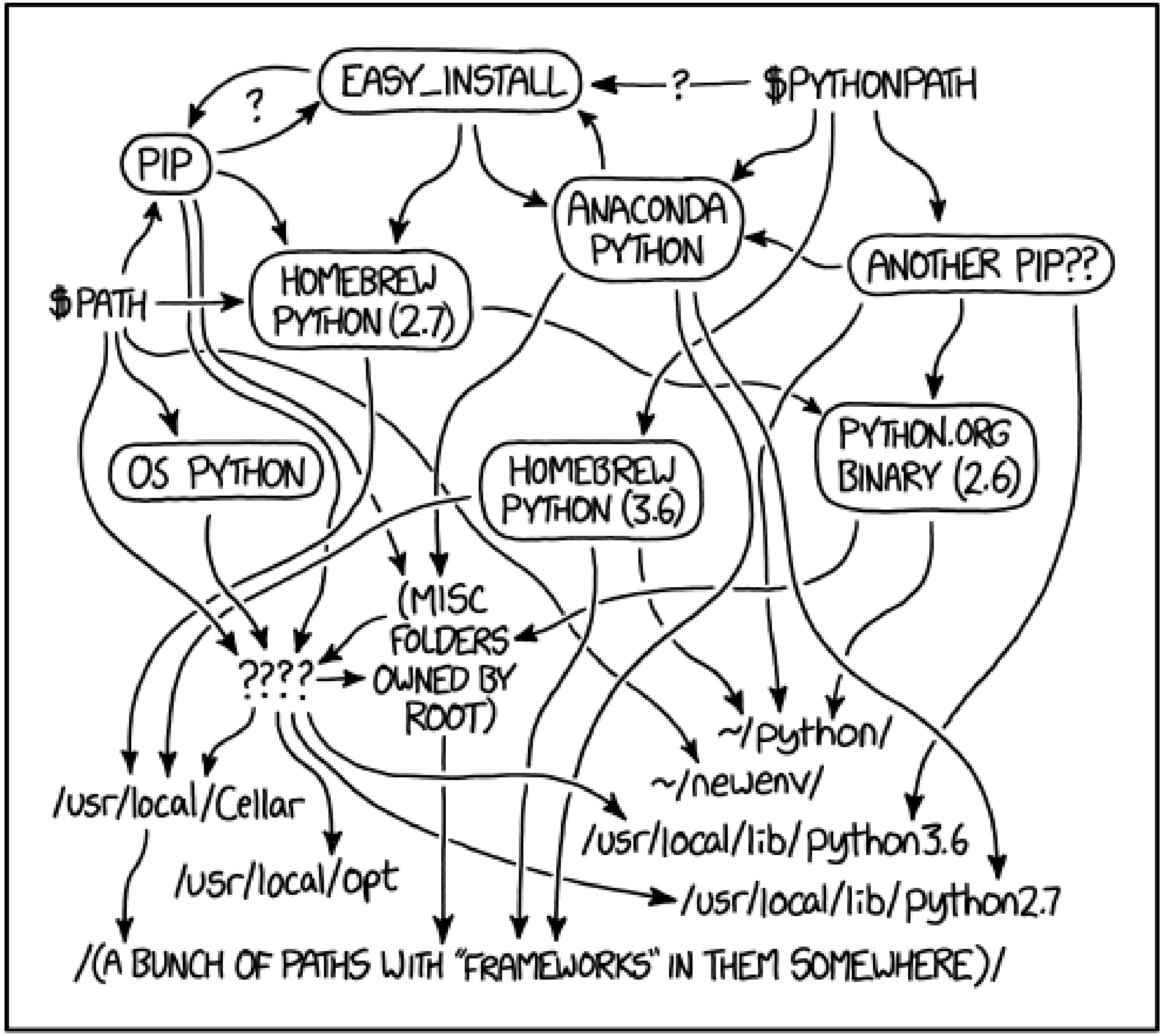
$$\begin{tikzcd}
\bar{x}_1 \arrow[r] \arrow[rr, bend right] & \bar{x}_2 \arrow[rr, bend left] & x_2 \arrow[r] & x_1
\end{tikzcd}$$

Граф 2-SAT

Если вершины в одной компоненте сильной связности, то мы их берем вместе, поэтому можно сделать конденсацию графа. Если пара оказалась в одной КСС, решения автоматически нет.

Иначе в имеющемся графе достаточно для каждой пары взять более позднюю вершину в порядке топсорта.

Мем



MY PYTHON ENVIRONMENT HAS BECOME SO DEGRADED THAT MY LAPTOP HAS BEEN DECLARED A SUPERFUND SITE.

Идея для пет-проекта

С помощью DAG и топологической сортировки можно составлять себе time schedule / study plan, если мы явно указали зависимости между темами или занятиями.

Для дальнейшего изучения

- Планировщики (scheduler-ы), графы зависимостей
- Все про деревья :)
- Ретроанализ (следующие слайды, не попавшие в лекцию)

Ретроанализ

Мы уже говорили про игры на графах, но не говорили про ничейные ситуации.

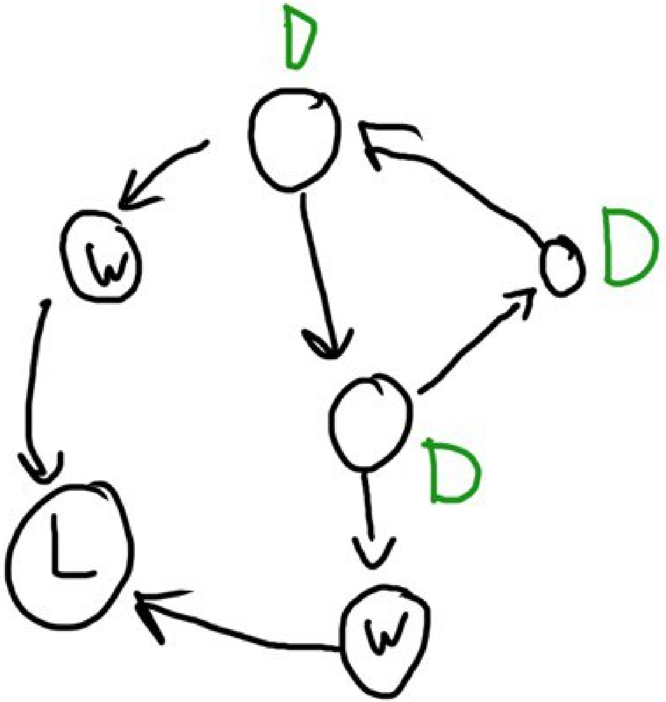
- Из W есть переход в L
- Из L все переходы в W

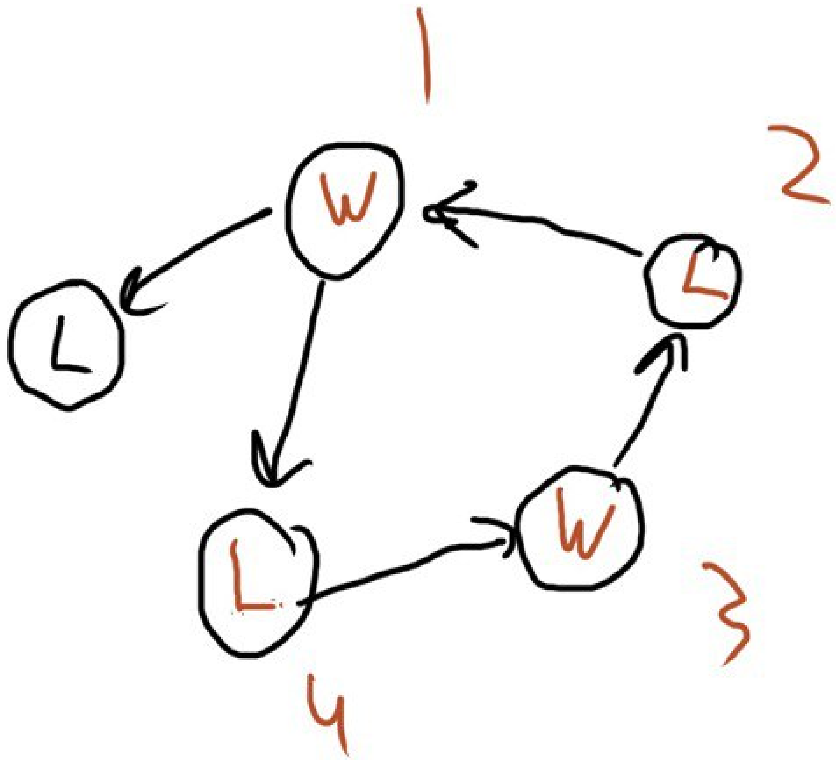
В случае циклов в графе игры может быть такое, что мы не можем определить для вершины ее состояние:

Например, из вершины v все переходы, кроме одного, ведут в W, но один оставшийся ведет в u, из которой достижима только v. Тогда v ничейная, потому что в ней можно бесконечно продолжать игру.

Контрпример с циклом

В общем случае цикл не всегда означает ничью - кому-то из игроков может быть выгодно "выйти" из цикла, чтобы выиграть. В ничейной ситуации ни один игрок не может выиграть и не хочет проигрывать.


D
W
D
D
L
W


1
W
2
L
L
W
3
L
4

Алгоритм

1. Проставляем базовые значения
2. Разворачиваем ребра
3. Если v проигрышная, то все прямые соседи v выигрышные
4. Если v выигрышная, то прямые соседи v игнорируют это ребро (счетчик количества непросмотренных ребер для всех u_i уменьшается)
5. Если v выигрышная и у соседа больше не осталось непросмотренных ребер, то он проигрышный.
6. Когда мы определяем состояние новой вершины, мы хотим попробовать узнать новые состояния уже для его прямых соседей.

Получается некоторая адаптация BFS. Те вершины, для которых остались непросмотренные ребра, окажутся ничейными.