1 Кратчайшие пути

Длина пути — сумма пометок, написанных на рёбрах графа (весов).

1.1 Графы без циклов отрицательного веса

Вершины на пути с наименьшей длиной не могут повторяться: иначе на пути есть цикл неотрицательного веса, который можно исключить из пути, уменьшив его длину.

Значит, если есть путь из одной вершины в другую, то есть вершинно-простой путь наименьшей длины.

1.2 Графы с циклами отрицательного веса

Если из вершины u есть путь в некоторый цикл отрицательного веса, и при этом есть некоторый путь из вершины на этом цикле в вершину v, то не существует кратчайшего пути из вершины u в v.

Так как по отрицательному циклу можно пройти сколь угодно большое число раз, предъявив путь меньше любого наперёд заданного числа.

Если такая ситуация невозможна, то существует вершинно-простой путь из u в v.

2 Алгоритм Форда-Беллмана

2.1 Постановка задачи

Алгоритм решает задачу *SSSP — Single Source Shortest Paths*, то есть находит кратчайшие пути от выделенной вершины s до всех остальных. Если кратчайшего пути в некоторую вершину v нет, то алгоритм определяет, какая ситуация произошла: путь существует, но на нём есть отрицательный цикл, или же пути из s в v не существует.

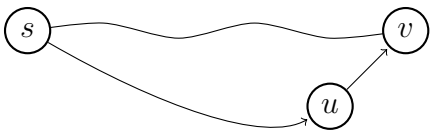
2.2 Релаксация ребра

$\text{dist}[v]$ — длина самого короткого пути из s в v, на данный момент найденная алгоритмом.

Рассмотрим ребро $u \to v$, имеющее вес w. Если $\text{dist}[v] > \text{dist}[u] + w$, то найденное расстояние $u \rightsquigarrow v$, равное $\text{dist}[v]$, можно улучшить на путь $s \rightsquigarrow u \to v$, проходящий через вершину u.

Чтобы восстановить найденный путь $s \rightsquigarrow v$, необходимо запомнить предпоследнюю вершину на пути: $\text{prev}[v] = u$.

```
def edge_relax(u, v, w):
  if dist[v] > dist[u] + w:
    dist[v] = dist[u] + w
    prev[u] = v
    return True
  return False
```



2.3 Основной алгоритм

2.3.1 Поиск кратчайших расстояний

Алгоритм Форда-Беллмана $(V-1)$ раз релаксирует все ребра в произвольном порядке.

```
dist = [+INF] * vn
dist[s] = 0
for i in range(vn - 1):
  for u, v, w in edges:
    relax_edge(u, v, w)
```

При дальнейших релаксациях рёбер, расстояние до вершин, до которых существует кратчайшее расстояние, обновляться не будет.

Очевидно, что асимптотическое время работы есть $O(VE)$.

2.3.2 Поиск множества вершин, достижимых из циклов отрицательного веса

После выполнения ещё одной релаксации всех рёбер на каждом отрицательном цикле обновится расстояние хотя бы до одной вершины.

Пометим их и из каждой запустим dfs. До всех вершин, которые пометил dfs, кратчайшего расстояния не существует.

```
updated = [False] * vn
for u, v, w in edges:
  if relax_edge(u, v, w):
    mark[v] = True

used = [False] * vn
for v in range(vn):
  if updated[v]:
    dfs(v)
```

2.3.3 Недостижимые из s вершины

Вершина v не достижима из s, тогда и только тогда, когда после работы алгоритма $\text{dist}[v] = +\infty$.

2.4 Тонкости реализации

1. Проверка в релаксации ребра `dist[v] > dist[u] + w` может сработать, если `dist[u] == +INF`, при условии, что $w < 0$. Расстояние до v обновится, и мы будем считать, что есть путь в v.

   Поэтому релаксировать ребро в графе с отрицательными рёбрами следует при условии: `dist[u] < +INF && dist[v] > dist[u] + w`.

2. Величины в массиве `dist` могут стать меньше `-INF` и переполнится. Это приведёт к некорректной работе алгоритма.

   Поэтому стоит проверять новое значение $\text{dist}[v]$ после релаксации и устанавливать в `-INF`, если нужно.

3. С учётом замечаний, релаксация ребра должна происходить так:

```
def edge_relax(u, v, w):
  if dist[u] < +INF and dist[v] > dist[u] + w:
    dist[v] = max(-INF, dist[u] + w)
    prev[u] = v
    return True
  return False
```

2.5 Доказательство корректности

Если существует кратчайший вершинно-простой путь, то на нём не более V вершин, а значит, не более $(V - 1)$ ребра.

Инвариант алгоритма — после k релаксаций всех рёбер графа, $\text{dist}[v]$ будет равно длине кратчайшего пути, на котором не более k рёбер.

Докажем это по индукции:

1. Очевидно, что это верно до первого шага: все вершины, кроме s, не достижимы по нулю рёбер.

   После первого шага, мы обновим только расстояние до смежных с s вершин, таким образом найдя кратчайшие расстояния из 1 ребра.

2. Кратчайшие пути из не более $(k-1)$ ребра к началу k-ой итерации мы уже нашли.

3. Любой путь из k рёбер, это путь из $(k-1)$ ребра и ещё одно ребро. Поскольку для каждой вершины мы релаксируем все входящие в неё рёбра, то после k-ой итерации, найдём кратчайшие пути из не более k рёбер.

3 Алгоритм Флойда-Уоршелла

3.1 Постановка задачи

Алгоритм решает задачу *APSP — All Pairs Shortest Paths*,
то есть находит кратчайшие пути между всеми парами вершин. Если кратчайшего пути между u и v нет, то алгоритм определяет, какая ситуация произошла: путь существует, но на нём есть отрицательный цикл, или же пути из u в v не существует.

3.2 Основной алгоритм

3.2.1 Поиск кратчайших расстояний

Находим кратчайшие расстояния с помощью динамического программирования. Кратчайшие пути вершинно-простые: каждая вершина используется не более одного раза.

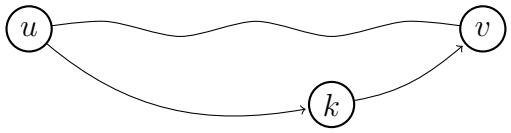
$\text{dist}_k[u][v]$ — длина кратчайшего пути из вершины u в вершину v по вершинам с номерами строго меньше k. Таким образом, состояние динамики — матрица $n \times n$.

Заметим, что матрица dist_0 — это матрица путей, не использующих промежуточные вершины, то есть просто матрица смежности.

База есть, осталось научиться переходить от матрицы dist_k к dist_{k+1}. Пути dist_k использовали вершины с номерами $0, 1, \ldots (k-1)$. В матрице dist_{k+1} разрешено использовать вершину с номером k.

Длина кратчайшего пути из u в v, использующего вершину k, равна $\text{dist}_k[u][k]+\text{dist}_k[k][v]$, если не использовать, то $\text{dist}_k[u][v]$. Таким образом:

$$\text{dist}_{k+1}[u][v] = \min\left(\text{dist}_k[u][v],\ \text{dist}_k[u][k] + \text{dist}_k[k][v]\right)$$




```
for k in range(vn):
  for u in range(vn):
    for v in range(vn):
      if dist[u][v] > dist[u][k] + dist[k][v]:
        dist[u][v] = dist[u][k] + dist[k][v]
```

Заметим, что можно хранить все величины в одной матрице dist, экономя память. Так как в момент обновления $\text{dist}[u][v] \leqslant \text{dist}_k[u][v]$ (пути покороче нам нравятся ещё больше).

3.2.2 Восстановление ответа

На каждом пути $u \rightsquigarrow v$ давайте хранить не только его величину, но и предпоследнюю вершину.

Изначально, при инициализации алгоритма, для каждого ребра $u \to v$ положим $\text{prev}[u][v] = u$, для остальных -1.

Предпоследняя вершина на пути $u \rightsquigarrow k \rightsquigarrow v$ такая же, как и на пути $k \rightsquigarrow v$. Поэтому, если при обновлении расстояния $\text{dist}[u][v]$ можно пересчитать $\text{prev}[u][v] = \text{prev}[k][v]$.

Имея предыдущую вершину на пути, легко восстановить и сам путь:

```
path = []
while v != u:
  path.append(v)
  v = prev[u][v]
path.append(u)
path.reverse()
```

Чтобы путь не переворачивать, можно хранить не $\text{prev}[u][v]$, а $\text{next}[u][v]$.

3.2.3 Тонкости реализации

Если есть отрицательные рёбра, то необходимо делать проверку: `dist[u][k] != +INF and dist[k][v] != +INF` (смотри замечания к алг. Форда-Беллмана).

А также стоит следить за тем, чтобы `dist[u][v] >= -INF`.

3.2.4 Графы с отрицательными рёбрами

Если есть отрицательные циклы и нет кратчайшего пути, то алгоритм находит длину какого-то пути, если он есть.

В этом случае вершины могут использоваться несколько раз (т.к. пути необязательно вершинно-простые), значит, восстановление пути не будет работать для таких путей.

На главной диагонали будут стоять длины не тривиальных циклов $\text{dist}[v][v]$ — длина пути из $v \rightsquigarrow v$. Неотрицательное число соответствует длине кратчайшего цикла, содержащего v, отрицательное — длине какого-то цикла отрицательной длины.

Из u в v нет кратчайшего пути тогда и только тогда, когда есть путь $u \rightsquigarrow k \rightsquigarrow v$, где k лежит на отрицательном цикле. Помечаем все такие пути:

```
for u in range(vn):
  for v in range(vn):
    for k in range(vn):
      if dist[k][k] < 0 and dist[u][k] < +INF \
                        and dist[k][v] < +INF:
        mark[u][v] = True
```

Если на таком пути пойти по массиву `prev`, восстанавливая путь, то мы зациклимся на цикле отрицательного веса. Так можно найти какой-то цикл отрицательного веса, если начать с вершины v, такой что $\text{dist}[v][v] < 0$.

4 Алгоритм Дейкстры

Очевидно! =)