1 Динамическое программирование

1.1 Общие принципы

1.1.1 Построение, оптимизация, комбинаторика

Формулировки задач динамического программирования чаще всего требует найти один из трёх результатов:

1. Построить некоторый объект с набором свойств. Например, найти в дереве путь, проходящий ровно по двум вершинам из выделенных.
2. Оптимизировать некоторый параметр. Например, найти наибольшую возрастающую подпоследовательность. Здесь подпоследовательность — это объект, а её длина оптимизируется.
3. Подсчитать количество способов построить некоторый объект. Например, найти количество чисел из цифр 0 и 1, делящихся на 3.

1.1.2 Подзадачи

Задачи, решаемые методом динамического программирования, имеют основную общую особенность. Задачу в них можно сформулировать так, что решение можно свести к решению аналогичных задач меньшего размера — *подзадач.*

(Оптимальность для подзадачи)

1.1.3 Состояния, переходы, база

1.1.4 ДП как перебор с запоминанием

1.1.5 Выделение параметров, избыточность

(Задача о трёх кучках монет)

(Добавить параметр, чтобы преобразовать оптимизацию в построение)

1.1.6 Восстановление ответа

1.2 ДП на префиксах

1.2.1 Кратчайший путь в ациклическом графе

Условие. Дан ориентированный граф без циклов. Найти длину кратчайшего пути от s до t.

Будем решать чуть более общую задачу — искать длину кратчайших путей от всех вершин до t ($\mathrm{dist}[v]$ — длина кратчайшего пути от v до u).

Топологически отсортируем граф. Массив topsort - перестановка вершин такая, что если если есть ребро $v \to u$, то u идет в topsort раньше, чем v. Найдем в этом массиве вершину t. Из вершин, идущих в topsort раньше, t недостижима. Поэтому для всех вершин v, идущих раньше $t, \mathrm{dist}[v] = +\infty$, а $\mathrm{dist}[t]$, очевидно, 0.

Будем считать $\text{dist}[v]$ динамикой по префиксу топологической сортировки: Предположим, что для всех вершин u, идущих в topsort-е раньше v $\text{dist}[u]$ посчитано корректно. (База - t и все вершины идущие раньше нее) Рассмотрим какой-нибудь путь от v до t. Рассмотрим w — вторую вершину на этом пути. w идет в topsort-е раньше, чем v (так как есть ребро $v \to w$), поэтому $\text{dist}[w]$ посчитано корректно. Если взять $\text{dist}[v] = \min\limits_{w:\exists v \to w} (\text{dist}[w] + \text{weight}(v \to w))$, то мы насчитаем правильный ответ для v, так как какая-то из ее соседей должна быть второй на пути из v в t (если же пути из v в t нет, то пути в t нет ни из одного соседа v)

Еще раз вкраце алгоритм: идем в порядке топологической сортировки графа и релаксируем ответ для текущей вершины по всем ребрам, из нее выходящим.

Рассмотрим немного другое решение этой задачи, использующее так называемую *рекурсию с меморизацией*.

Примерный код:

```
dist = [INF] * N
counted = [False] * N
dist[t] = 0
counted[t] = True

def dfs(v)
    if not counted[v]:
        counted[v] = True
        for (u, weight) in g[v]:
            dist[v] = min(dist[v], dist[u] + weight)
    return dist[v]
```

Разберемся, что тут происходит.

Утверждение: $\text{dfs}(v)$ возвращает корректное кратчайшее расстояние от v до t.

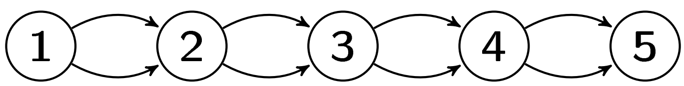
Докажем его индукцией по времени выхода из вершины.

База: если из вершины не выходит ни одного ребра, то это либо t, и тогда dfs вернет 0, что и требовалось, либо INF в противном случае, что тоже верно

Переход: У всех соседей вершины v время выхода не больше, чем у v (Свойство dfs на графе без циклов, то есть без обратных ребер). Поэтому, применяя рассуждения предыдущего решения ($\text{dist}[v] = \min\limits_{w:\exists v \to w} (\text{dist}[w] + \text{weight}(v \to w))$) получаем корректность алгоритма.

Алгоритм работает не более чем $\mathcal{O}(V + E)$, так как каждую вершину и каждое ребро мы просмотрим не более одного раза (благодаря массиву counted)

Если бы мы убрали массив counted, и поставили условие `if v == t: return 0` то код бы тоже работал, но, возможно, за очень долгое время (например, на бамбуке с раздвоенными ребрами он может работать $\mathcal{O}(2^V)$)



Если $s = 1$ и $t = 5$ dfs вызовется 31 раз

Но мы пользуемся тем, что если мы один раз посчитали ответ для вершины, его можно запомнить и в дальнейшем не пересчитывать. Такой прием и называется *меморизацией*.

При реализации алгоритма не забывайте, что $\infty + weight = \infty$ даже при отрицательных $weight$

1.2.2 НВП

1.2.3 НОП

1.3 ДП по цифрам числа

1.3.1 Количество чисел от 0 до n с суммой цифр k

Условие. Дано число N длины n и k. Нужно посчитать количество чисел от 0 до N

1.4 ДП на подотрезках

1.4.1 Оптимальное перемножение матриц

1.4.2 Наибольшая подпоследовательность-палиндром, подматрица-палиндром

1.4.3 Казино

1.5 ДП на поддеревьях

1.5.1 Взвешенное паросочетание в дереве

Ребро e *инцидентно* вершине v и наоборот, вершина v *инцидентна* ребру e, если v является одним из концов e.

Условие. Дано взвешенное дерево. Найти паросочетание (то есть такой набор ребер, что каждой вершине инцидентно не более одного ребра из нашего набора) максимального суммарного веса.

Подвесим дерево за какую-нибудь вершину. Будем считать динамику $\text{dp}[v][0]$ — ответ для поддерева вершины v, если вершина v не инцидентна ни одному ребру из набора, и $\text{dp}[v][1]$ — ответ для поддерева вершины v без дополнительных ограничений.

Примерный код:

```python
dp = [[0, 0] for i in range(N)]

def dfs(v):
     for (u, weight) in g[v]:
        dfs(u)
        dp[v][1] = max(dp[v][1] + dp[u][1], dp[v][0] + dp[u][0] + weight)
        dp[v][0] = dp[v][0] + dp[u][1]
```

Подразумевается, что в $\text{g}[v]$ хранится список смежности вершины v без родителя.

Разберемся, почему этот код работает: **Утверждение:** на h-м шаге внешнего цикла (4 строчка) в dp[v][0] и dp[v][1] хранится ответ, для начала поддерева вершины v из нее самой и первых h поддеревьев (нумерация с 1)

Докажем его. При h $=$ 0 утверждение очевидно. Пусть утверждение верно после h шагов, покажем, что оно верно и после $h+1$ шага. Мы можем либо брать в наше паросочетание ребро в очередного ребенка u, либо не брать. Заметим, что оставшиеся ребра либо лежат в начале поддереве v (из нее и первых h поддеревьев), либо в поддереве u.

Разберем случай dp[v][1], то есть когда мы можем использовать вершину v в паросочетании:

- Пусть мы берем ребро $v \to u$. Рассмотрим оставшееся паросочетание. В нем ни v, ни u не должны были быть использованы, а сумма ребер должна быть максимальной. Поэтому ответ для этого случая равен dp[v][0] $+$ dp[u][0] $+$ weight.

- Если же мы не берем ребро $v \to u$, то мы свободны брать и не брать как u, так и v. Этому случаю отвечает величина dp[v][1] $+$ dp[u][1]

В случае dp[v][0], то есть когда мы не можем использовать вершину v брать ребро $v \to u$ уже нельзя. Остается набрать из начала поддерева v (из первых h поддеревьев) и из поддерева u ребер по максимуму. Это и выражает формула dp[v][0] $+$ dp[u][1]

Время работы, очевидно, $\mathcal{O}(N)$

1.5.2 Выделение поддерева размера k

Условие. Дано дерево на N вершинах и число K. Найдите минимальное количество ребер, которое надо удалить из дерева, чтобы одна из компонент оказалась размера ровно K. $\mathcal{O}(N^2)$

Подвесим дерево за какую-нибудь вершину. Будем считать динамику dp[v][k] — минимальное количество ребер, при удалении которого поддерево v распадется на такие компоненты, что содержащая v будет состоять из k вершин.

Примерный код:

```
dp = [[INF] * (N + 1) for i in range(N)]
size = [0] * N
def dfs(v):
    size[v] = 1;
    dp[v][1] = len(g[v])

    for u in g[v]:
        dfs(u)
        for i in range(len(size[v]), 0, -1):
            for j in range(1, len(size[u]) + 1):
                dp[v][i + j] = min(dp[v][i + j], dp[u][j] + dp[v][i] - 1)
            size[v] += size[u]
```

Подразумевается, что в g[v] хранится список смежности вершины v без родителя.

Разберемся почему этот код работает.

Утверждение: на h-м шаге внешнего цикла (7 строчка) в dp[v][k] хранится ответ, если в компоненту с вершиной v разрешается брать лишь вершины из первых h поддеревьев (нумерация с 1)

Докажем его. При h = 0 утверждение очевидно. Пусть утверждение верно после h шагов, покажем, что оно верно и после $h+1$ шага. В нашу компоненту мы можем либо брать вершины из h-го поддерева, либо нет. Если мы не берем, то, по предположению индрукции ответ dp[v][k]. Если же мы берем, то посмотрим, сколько вершин мы взяли из предыдущих поддеревьев (за это отвечает переменная i), а сколько - из $(h+1)$-го поддерева (за это отвечает j). ну и платим мы за это соответственно dp[u][j] + dp[v][i] - 1 (-1 так как раньше мы удаляли ребро между v и ее $(h+1)$-м ребенком, а теперь нет). Поскольку i и j пробегают все возможные значения, то ответ посчитается корректно.

Утверждение: dfs(v) работает за $\mathcal{O}(\text{size}[v]^2)$

Докажем по индукции по размеру поддерева. Пусть ω_h - размер поддерева h-го ребенка v. Обозначим за z количество детей v В цикле (7) мы вызовем dfs от всех детей (это будет $\sum(c \cdot \omega_h^2)$ по предположению индукции. Возьмем $c > 2$). На h-м шаге size[v] $= (1 + \omega_1 + \cdots + \omega_{h-1})$ соответственно на h-м шаге цикл (9) выполнит суммарно $\omega_{h+1} \cdot (\sum\limits_{j \leqslant h}(\omega_h) + 1)$ операций.

$$\sum_{h=0}^{z-1}(\omega_{h+1} \cdot (\sum_{j \leqslant h}(\omega_h) + 1)) = \sum_{1 \leqslant i \leqslant j \leqslant z}(w_i \cdot w_j) + \sum_{i \leqslant z} w_i$$

$$\sum_{i \leqslant z}(cw_i^2) \sum_{1 \leqslant i < j \leqslant z}(w_i \cdot w_j) + \sum_{i \leqslant z} w_i \overset{c>2}{\leqslant} c \cdot (\sum_{i \leqslant z} w_i)^2$$

Что и требовалось доказать.

Таким образом, dfs($root$) работает за $\mathcal{O}(N^2)$. Как же найти ответ на исходную задачу? Так как какая-то вершина в полученной компоненту будет иметь минимальную глубину, то достаточно запустить dfs($root$) и потом выбрать минимум из dp[v][K] + 1 по всем v кроме $root$ и dp[$root$][K] (так как если мы не корень, нужно еще обрезать ребро в родителя).

1.5.3 Задача о разделении дерева