

Лекция 8. Динамическое программирование (по подмножествам, по профилю).

1. Введение в ДП на подмножествах

Динамическое программирование на подмножествах – это метод решения комбинаторных задач, где состояние включает в себя выбор элементов из некоторого множества. Если множество содержит n элементов, то всего существует 2^n возможных подмножеств. Задачи, которые кажутся переборными (с экспоненциальной сложностью $O(2^n)$), часто оказываются решаемыми с помощью ДП за $O(2^n \cdot \text{poly}(n))$, что приемлемо для $n \sim 20$.

Ключевая идея: представлять подмножества в виде **битовых масок** – целых чисел, где i -й бит равен 1, если i -й элемент включён в подмножество, и 0 в противном случае.

Примеры задач:

- Задача коммивояжёра (TSP)
- Раскраска графа
- Покрытие множества
- Распределение задач по процессорам

2. Битовая арифметика для работы с подмножествами

Пусть у нас есть множество из n элементов, пронумерованных от 0 до $n - 1$.

Базовые операции:

```
n = 5
mask = 0b11010 # Множество {1, 3, 4}

# Проверка наличия элемента i
i = 3
has_element = (mask >> i) & 1 == 1 # True

# Добавление элемента i
mask |= (1 << i)

# Удаление элемента i
mask &= ~(1 << i)

# Переключение элемента i
mask ^= (1 << i)

# Объединение множеств
mask1 = 0b11010
mask2 = 0b01101
union = mask1 | mask2 # 0b11111

# Пересечение множеств
intersection = mask1 & mask2 # 0b01000
```

```

# Дополнение
complement = ((1 << n) - 1) ^ mask

# Все подмножества множества mask
submask = mask
while submask > 0:
    # Обработать submask
    submask = (submask - 1) & mask

```

3. Пример 1. Задача коммивояжёра (TSP).

Постановка. Дан взвешенный граф из n вершин (городов). Необходимо найти гамильтонов цикл минимального веса (посетить все города по одному разу и вернуться в начальный).

Решение ДП:

Состояние: $dp[mask][i]$ – минимальная стоимость посещения всех городов из множества $mask$, заканчивая в городе i .

База: $dp[1 << i][i] = 0$ (если мы начинаем и заканчиваем в одном городе без перемещений).

Переход. $dp[mask][i] = \min(dp[mask][i], dp[mask_without_j][j] + dist[j][i])$ для всех j из $mask$, где есть ребро $j \rightarrow i$.

Ответ: $\min(dp[(1 << n) - 1][i] + dist[i][0])$ для всех i (вернуться в начальный город).

Реализация:

```

def tsp(dist):
    """
    Решает задачу коммивояжёра методом динамического программирования.

    Args:
        dist: матрица расстояний  $n \times n$ , где  $dist[i][j]$  - расстояние от  $i$  до  $j$ 

    Returns:
        Минимальная длина гамильтонова цикла
    """

    n = len(dist)
    # Инициализация DP таблицы:  $dp[mask][i]$  - минимальная стоимость посетить
    # все города из  $mask$ , закончив в городе  $i$ 
    dp = [[float('inf')] * n for _ in range(1 << n)]

    # Базовый случай: начальные вершины
    # Если мы только в одном городе, стоимость 0
    for i in range(n):
        dp[1 << i][i] = 0

    # Заполнение DP таблицы
    # Перебираем все возможные маски (подмножества городов)
    for mask in range(1 << n):
        # Для каждого города  $i$  в текущем множестве
        for i in range(n):
            # Если город  $i$  не входит в текущее множество, пропускаем

```

```

        if not (mask >> i) & 1:
            continue
        # Пытаемся прийти в город j из города i
        for j in range(n):
            # Если город j уже посещён в этом множестве, пропускаем
            if (mask >> j) & 1:
                continue
            # Новое множество с добавленным городом j
            new_mask = mask | (1 << j)
            # Обновляем стоимость для нового множества и города j
            dp[new_mask][j] = min(dp[new_mask][j], dp[mask][i] +
dist[i][j])

        # Поиск минимального цикла (вернуться в начальный город)
ans = float('inf')
full_mask = (1 << n) - 1 # Мaska всех городов
for i in range(n):
    # Стоимость посетить все города + вернуться в начальный
    ans = min(ans, dp[full_mask][i] + dist[i][0])

return ans

# Пример использования
dist = [
    [0, 10, 15, 20],
    [10, 0, 35, 25],
    [15, 35, 0, 30],
    [20, 25, 30, 0]
]
print(f"Минимальная длина цикла: {tsp(dist)}") # Вывод: 80
Сложность:  $O(2^n \cdot n^2)$ , что приемлемо для  $n \leq 20$ .

```

4. Пример 2. Раскраска графа.

Постановка. Дан граф из n вершин. Найти минимальное количество цветов, необходимое для раскраски вершин так, чтобы смежные вершины имели разные цвета.

Решение ДП:

Состояние: $dp[mask]$ – минимальное число цветов, необходимых для раскраски множества вершин $mask$.

База: $dp[0] = 0$ (пустое множество не требует цветов).

Переход. Перебираем все независимые подмножества S множества $mask$ (где нет смежных вершин) и пытаемся раскрасить их одним цветом:

$dp[mask] = \min(dp[mask], dp[mask \wedge S] + 1)$.

Реализация:

```

def graph_coloring(adj):
    """
    Находит хроматическое число графа (минимальное количество цветов).

```

Args:

adj: матрица смежности $n \times n$

Returns:

Хроматическое число графа

"""

```
n = len(adj)
# Предвычисление независимых множеств
# is_independent[mask] = True, если множество mask независимое
is_independent = [True] * (1 << n)

# Проверяем каждое множество на независимость
for mask in range(1 << n):
    for i in range(n):
        # Если вершина i в множестве
        if (mask >> i) & 1:
            for j in range(n):
                # Если есть ребро между i и j, и j тоже в множестве
                if adj[i][j] and (mask >> j) & 1:
                    is_independent[mask] = False
                    break
            if not is_independent[mask]:
                break

# Динамическое программирование
# dp[mask] - минимальное число цветов для раскраски множества mask
dp = [float('inf')] * (1 << n)
dp[0] = 0 # Пустое множество требует 0 цветов

# Перебираем все множества
for mask in range(1 << n):
    # Перебираем все подмножества mask
    submask = mask
    while submask > 0:
        # Если подмножество независимое, можем раскрасить его одним цветом
        if is_independent[submask]:
            dp[mask] = min(dp[mask], dp[mask ^ submask] + 1)
            submask = (submask - 1) & mask

return dp[(1 << n) - 1] # Ответ для полного множества вершин

# Пример использования
adj = [
    [0, 1, 1, 0],
    [1, 0, 1, 1],
    [1, 1, 0, 1],
    [0, 1, 1, 0]
]
print(f"Хроматическое число: {graph_coloring(adj)}") # Вывод: 3
Сложность:  $O(3^n)$  (по теореме о сумме биномиальных коэффициентов).
```

5. Динамическое программирование по профилю

Динамическое программирование по профилю применяется для задач замощения и других задач на сетках, где состояние описывает «границу» между обработанной и необработанной частями.

Ключевая идея: разбить сетку на слои и хранить состояние последнего слоя (профиль), которое влияет на следующие слои.

6. Пример 3. Замощение доминошками.

Постановка. Данна сетка $n \times m$. Найти количество способов замостить её доминошками 1×2 и 2×1 .

Решение ДП по профилю:

Состояние: $dp[i][mask]$ – количество способов замостить первые i столбцов, где $mask$ описывает профиль – какие клетки i -го столбца заняты доминошками, выходящими в $(i+1)$ -й столбец.

Переход. Перебираем все валидные способы размещения доминошек в $(i+1)$ -м столбце, которые совместимы с профилем $mask$.

Реализация:

```
def domino_tiling(n, m):
    """
    Считает количество способов замостить сетку n x m доминошками 1x2 и 2x1.

    Args:
        n, m: размеры сетки

    Returns:
        Количество способов замощения
    """
    # Для эффективности делаем n >= m (меньшая размерность в битовой маске)
    if n < m:
        n, m = m, n

    # dp[i][mask] - количество способов замостить первые i столбцов
    # mask описывает профиль - какие клетки i-го столбца заняты доминошками,
    # выходящими в (i+1)-й столбец
    dp = [[0] * (1 << m) for _ in range(n + 1)]
    dp[0][0] = 1 # Базовый случай: 0 столбцов замощено

    # Обрабатываем каждый столбец
    for i in range(n):
        for mask in range(1 << m):
            if dp[i][mask] == 0:
                continue # Пропускаем невозможные состояния

    # Рекурсивная функция для генерации следующих состояний
    def dfs(col, next_mask):
        """
        Рекурсивно генерирует все валидные замощения для следующего
        столбца.
        """
        pass
```

```

Args:
    col: текущая обрабатываемая строка в столбце
    next_mask: формируемая маска для следующего столбца
"""
if col == m:
    # Достигли конца столбца - обновляем DP
    dp[i + 1][next_mask] += dp[i][mask]
    return

# Если текущая клетка уже занята (вертикальной доминошкой из
предыдущего столбца)
if (mask >> col) & 1:
    # Просто переходим к следующей клетке
    dfs(col + 1, next_mask)
else:
    # Попробовать горизонтальную доминошку (занимает 2 клетки
в одном столбце)
    if col + 1 < m and not (mask >> (col + 1)) & 1:
        dfs(col + 2, next_mask)

    # Попробовать вертикальную доминошку (занимает клетку в
следующем столбце)
    dfs(col + 1, next_mask | (1 << col))

# Запускаем рекурсивный перебор для текущего состояния
dfs(0, 0)

return dp[n][0] # Ответ: все столбцы замощены, ничего не выпирает

# Пример использования
n, m = 3, 3
print(f"Количество способов замощения {n}x{m}: {domino_tiling(n, m)}") # Вы-
вод: 0

n, m = 2, 3
print(f"Количество способов замощения {n}x{m}: {domino_tiling(n, m)}") # Вы-
вод: 3

```

Сложность: $O(n \cdot m \cdot 2^m)$, что эффективно при небольших m .

7. Оптимизации

7.1. Хранение только двух слоёв

Для экономии памяти в ДП по профилю часто хранят только текущий и предыдущий слой:

```

def domino_tiling_optimized(n, m):
    if n < m:
        n, m = m, n

    dp_prev = [0] * (1 << m)
    dp_prev[0] = 1

```

```

for i in range(n):
    dp_curr = [0] * (1 << m)
    for mask in range(1 << m):
        if dp_prev[mask] == 0:
            continue

        # Аналогичная рекурсивная функция
        def dfs(col, next_mask):
            if col == m:
                dp_curr[next_mask] += dp_prev[mask]
                return

            if (mask >> col) & 1:
                dfs(col + 1, next_mask)
            else:
                if col + 1 < m and not (mask >> (col + 1)) & 1:
                    dfs(col + 2, next_mask)
                dfs(col + 1, next_mask | (1 << col))

        dfs(0, 0)
    dp_prev = dp_curr

return dp_prev[0]

```

7.2. Предвычисление переходов

Для ускорения можно предвычислить все возможные переходы между профилями:

```

def precompute_transitions(m):
    transitions = [[] for _ in range(1 << m)]

    for mask in range(1 << m):
        def dfs(col, next_mask):
            if col == m:
                transitions[mask].append(next_mask)
                return

            if (mask >> col) & 1:
                dfs(col + 1, next_mask)
            else:
                if col + 1 < m and not (mask >> (col + 1)) & 1:
                    dfs(col + 2, next_mask)
                dfs(col + 1, next_mask | (1 << col))

        dfs(0, 0)

    return transitions

def domino_tiling_fast(n, m):
    if n < m:
        n, m = m, n

    transitions = precompute_transitions(m)

```

```

dp_prev = [0] * (1 << m)
dp_prev[0] = 1

for i in range(n):
    dp_curr = [0] * (1 << m)
    for mask in range(1 << m):
        if dp_prev[mask] == 0:
            continue
        for next_mask in transitions[mask]:
            dp_curr[next_mask] += dp_prev[mask]
    dp_prev = dp_curr

return dp_prev[0]

```

8. Сравнение методов

Метод	Применение	Сложность	Память	Особенности
ДП на подмножествах	Комбинаторные задачи	$O(2^n \cdot \text{poly}(n))$	$O(2^n)$	Универсальный, но требует много памяти
ДП по профилю	Замощения, сетки	$O(n \cdot 2^m)$	$O(2^m)$	Специализированный, эффективен при небольших m