Основные алгоритмы, комментарии по листкам 12-13 (Динамическое программирование I/II)

Шибаев Иннокентий

May 2, 2021

1 По поводу теории

1.1 Обзор изученных алгоритмов на графах

Перед собственно обсуждением динамического программирования хотелось бы обсудить то какие в нашем курсе были рассмотрены алгоритмы на графах. Как и в случае сортировок, приведем сводную таблицу 1 с основной информацией.

Примечание 1.1. Для алгоритмов на графах нам нужно хранить граф. Два основных способа: *матрица смежности* и *списки смежности*. Для первого надо хранить всю матрицу размера $O\left(|V|^2\right)$, зато можно быстро (за $O(1)$) отвечать на вопросы вида «есть ли ребро (u, v), и какой у него вес (если граф взвешенный)». С другой стороны в таком случае сложно перебирать ребра исходящие из вершины – нам придется перебрать всю строку матрицы (представьте, к примеру, что мы говорим о графе Интернета, где вершины – страницы, а ребра – ссылки между ними. Очевидно что число вершин связанных с данной намного меньше всего числа вершин).

Поэтому если в задаче нам надо перебирать только ребра исходящие из вершин то лучше использовать списки смежности – а это всего $O(|V| + |E|)$ (если считать что мы храним это как массив списков). Конечно в таком случае проверять наличие ребра сложнее.

А еще если вы уже потратили память на матрицу смежности, то наверное вам хватит дополнительно и списки смежности поддерживать – и тогда можно пользоваться преимуществами обоих структур.

Для взвешенного графа, понятно, можно хранить еще и вес соответствующих ребер, память от этого разве что в константу раз увеличивается.

Примечание 1.2. Алгоритм поиска в глубину сам по себе не то чтобы интересен. Это просто способ обхода графов, соответствующий тому как мы могли бы обходить лабиринт. Но этот алгоритм лежит в основе многих других, к примеру

- Поиск цикла в графе (если в процессе обхода встречаем серую вершину – есть цикл)
- Поиск Эйлерова цикла (опять же – встречаем серую вершину – есть цикл – вырезаем его – дальше надо склеить циклы)
- Классификация ребер (для этого в процессе обхода заполняем массивы $d[v]/f[v]$ времен входа (начала обработки) и выхода (конца обработки) каждой вершины)
- Топологическая сортировка (упорядочить вершины в порядке времен выхода из них)
- Разбиение на компоненты сильной связности (топологическая сортировка, инвертировать ребра и DFS)

И все эти вещи можно делать за $O(|V| + |E|)$ – за линию от длины входа!

Таблица 1: Общая информация об алгоритмах на графах в этом курсе

Алгоритм	Сложность по времени	Сложность по памяти	Суть	Зачем применяется
DFS, Поиск в глубину	$O(\lvert V\rvert + \lvert E\rvert)$	Список смежности: $O(\lvert V\rvert + \lvert E\rvert)$; Покраска вершин / массивы $d[v]$ и $f[v]$ / массив предков $p[v]$: $O(\lvert V\rvert)$	Заходим в непомеченную (белую) вершину, помечаем ее серым и переходим в непомеченного потомка. Если таких нет – помечаем саму вершину как обработанную (черным) и возвращаемся	Много для чего, см. примечание 1.2
BFS, Поиск в ширину	$O(\lvert V\rvert + \lvert E\rvert)$	Список смежности: $O(\lvert V\rvert + \lvert E\rvert)$; Очередь вершин для обработки и сами расстояния до вершин: $O(\lvert V\rvert)$	Извлекаем вершину из очереди, смотрим на соседей, тех что еще не были в очереди – добавляем с расстоянием на 1 большим	Поиск кратчайших путей от вершины s до остальных в графе с **единичными** весами на ребрах
Алгоритм Дейкстры	$O\left(\lvert V\rvert^2 + \lvert E\rvert\right)$ или $O(\lvert E\rvert \log \lvert V\rvert)$, см. примечание 1.3	Список смежности: $O(\lvert V\rvert + \lvert E\rvert)$; Массив или очередь с приоритетами для вершин и расстояний до них: $O(\lvert V\rvert)$	Смотрим на вершину с минимальным до нее расстоянием, смотрим на соседей, релаксируем расстояния если получается	Поиск кратчайших путей от вершины s до остальных в графе с **неотрицательными** весами на ребрах
Алгоритм Форда-Беллмана	$O(\lvert V\rvert\lvert E\rvert)$	Список смежности: $O(\lvert V\rvert + \lvert E\rvert)$; Массив расстояний: $O(\lvert V\rvert)$	$\lvert V\rvert - 1$ раз проходим по всем ребрам и пытаемся выполнить релаксации	Поиск кратчайших путей от вершины s до остальных во взвешенном графе
Алгоритм Флойда-Уоршелла	$O\left(\lvert V\rvert^3\right)$	Матрица смежности: $O\left(\lvert V\rvert^2\right)$; Массив расстояний для всех пар вершин: $O\left(\lvert V\rvert^2\right)$	На k-м шаге проходим по всем парам i, j вершин сравнивая $d[i,j]$ и $d[i,k] + d[k,j]$ и беря меньшее	Поиск кратчайших путей между всеми парами вершин во взвешенном графе
Алгоритм Краскала	$O(\lvert E\rvert \log \lvert V\rvert)$	Отсортированный список ребер – $O(\lvert E\rvert)$; DSU для ответа на вопрос о принадлежности одной компоненте: $O(\lvert V\rvert)$	Сливаем ближайшие компоненты пока не получится дерево. Реализация: сортируем ребра по возрастанию весов, добавляем ребро если оно не образует цикл с уже взятыми	Построение минимального остовного дерева
Алгоритм Прима	Как у алгоритма Дейкстры: $O\left(\lvert V\rvert^2 + \lvert E\rvert\right)$ или $O(\lvert E\rvert \log \lvert V\rvert)$	Как у алгоритма Дейкстры	Наращиваем компоненту начиная с какой-то вершины, каждый раз добавляем вершину до которой наименьшее расстояние от компоненты. Реализация: алгоритм Дейкстры + меняем условие релаксации	Построение минимального остовного дерева

Примечание 1.3. Сложность алгоритма Дейкстры зависит от реализации. При использовании простого массива расстояний поиск необработанной вершины с минимальным расстоянием до нее можно проводить за $O(|V|)$, а релаксации делаются за $O(1)$, поэтому получается сложность $O\left(|V|^2 + |E|\right)$.

Другой вариант – реализовать все через очередь с приоритетами. Тогда обе операции выше будут выполнятся за $\log |V|$, и получится сложность $O\left(|E| \log |V|\right)$.

Что выбирать – зависит от того сколько ребер в графе. Если граф разрежен то второй вариант быстрее.

Примечание 1.4. Важно, что в задачах поиска кратчайших путей всегда подразумевается что **отрицательных циклов нет** – иначе кратчайшие пути просто не определены.

Отдельный случай это задача поиска отрицательного цикла в графе. Это можно делать как раз алгоритмом Форда-Беллмана, запустив его еще на одну итерацию – если происходит хотя бы одна релаксация то цикл есть.

Примечание 1.5. Для алгоритмов поиска кратчайших путей также важно восстановить сам путь. Для BFS, алгоритма Дейкстры и алгоритма Форда-Беллмана это делается довольно просто – достаточно поддерживать массив предков $p[v]$ вершин – тогда при успешной релаксации по ребру (u, v) надо записать $p[v] = u$.

Для алгоритма Флойда-Уоршелла эта задача становится немного более сложной, но мы можем поддерживать двумерный массив $p[i, j]$ и в случае когда происходит релаксация по вершине k (т.е. $d[i, j] > d[i, k] + d[k, j]$) писать $p[i, j] = k$. Тогда кратчайший путь будет иметь вид уже $i \to k \to j$, и, рекурсивно вычислив кратчайшие пути $i \to k$ и $k \to j$ мы сможем восстановить путь.

1.2 О динамическом программировании

До примеров здесь обсуждается в основном вопрос о том что вообще считать задачей/алгоритмом динамического программирования (ДП), и это вполне можно пропустить, т.к. примеры в этом разделе куда лучше отвечают на этот вопрос.

ДП – это когда ваша задача может быть решена через оптимальные решение каких-то подзадач. Т.е. если у вас есть задача, вы можете разбить ее (получить ее решение из) некоторого кол-ва задач меньшего размера, и те задачи вы тоже можете тем же образом разбивать рекурсивно – перед вами ДП.

Вот в таком виде под ДП можно и сортировки подогнать – разбитие на подзадачи есть, восстановление ответа по ответам к подзадачам (вспомните, к примеру, $MergeSort$) – есть.

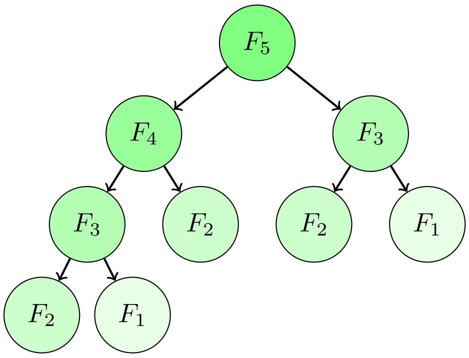
Но чаще все же под ДП имеется в виду случай когда наблюдается комбинация следующих двух условий:

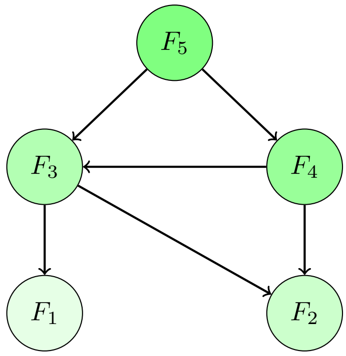
- Оптимальность структуры задачи, в том смысле что решение исходной задачи можно сконструировать из *оптимальных* решений ее подзадач, иными словами в структуре задачи можно применять в каком-то смысле жадные алгоритмы;
- Перекрытие подзадач. Идея в том что при рекурсивном вызове на подзадачах мы можем столкнуться с тем что решаем ту же задачу что уже решили в другой ветке. Поэтому часто возникает запоминание ответов для подзадач, чтобы не приходилось решать их заново.

Теперь пойдем по примерам, и будем смотреть на то как эти свойства проявляются в них.

1.2.1 Числа Фибоначчи

У нас есть рекуррентная зависимость: $F_n = F_{n-1} + F_{n-2}$, и начальные условия: $F_1 = 1, F_2 = 1$. Казалось бы, в чем проблема запустить рекурсивный алгоритм и вычислить F_n? Посмотрим на граф вычисления F_5 слева:





Можно заметить, что в ходе рекурсивного вычисления мы дважды вычисляем F_3. Вот тут и проявляется второе свойство – перекрытие подзадач. Орграф зависимостей (справа) – это ациклический ориентированны граф, и поднимаясь в нем от вершин в которых мы ответ знаем, мы получим ответ для F_5 – при этом повторных вычислений происходить не будет.

То что мы здесь сделали – и есть запоминание. Мы нашли в графе повторы, и переставили стрелки, иными словами – запомнили ответ чтобы не вычислять его в следующий раз.

И в этом примерно все ДП – какие-то повторяющиеся в рекурсивном вычислении подзадачи мы запоминаем, и при повторном запросе сразу выдаем ответ, вместо их повторного решения.

В данном конкретном случае, конечно, осмысленно поддерживать вообще только два последних значения – F_{n-2} и F_{n-1}. Но это уже пошли вопросы об оптимизации потребления памяти.

1.2.2 Длина максимальной возрастающей подпоследовательности

Есть последовательность чисел $x_1, \ldots, x_n$. Надо найти в ней возрастающую подпоследовательность (а.к.а. последовательность индексов $1 \leqslant i_1 < i_2 < \ldots < i_r \leqslant n$ таких что $x_{i_k} > x_{i_{k-1}}$) наибольшей длины (и вернуть ее длину).

Давайте рассуждать как к этой задаче применить ДП. Входом в задаче является последовательность чисел $x_1, \ldots, x_n$. Выходом – число, длина наибольшей возрастающей подпоследовательности, обозначим его $A(x_1, \ldots, x_n)$. Надо понять, как нам перейти к подзадачам.

Посмотрим на элемент x_n – последний в последовательности. Есть два варианта:

1. x_n не входит в самую длинную возрастающую подпоследовательность. Это прекрасный случай, т.к. тогда ответ это просто $A(x_1, \ldots, x_{n-1})$

2. x_n входит в такую последовательность. Это значит, что в последовательности $x_1, \ldots, x_{n-1}$ есть какая-то длинная подпоследовательность, последний элемент которой меньше чем x_n – присоединяя к ней x_n мы получаем ответ.

Если бы мы умели решать оба случая, то ответ был бы просто максимум из этих двух вариантов (варианта где x_n не входит в ответ, и варианта где он входит). Первый вариант это просто та же задача но с меньшей длиной последовательности. Но что делать со вторым пунктом? Если бы мы умели как-то находить какую последовательность можно подсоединить к x_n задача была бы решена. Такая последовательность должна заканчиваться каким-то $x_i < x_n$. Т.е. если бы мы для каждого $x_i < x_n$ знали самую длинную последовательность с концом в x_i – мы могли бы просто пройтись по ним и присоединить x_n к длиннейшей из них!

И вот здесь мы совершили переход уже к немного другой задаче – поиску длины наибольшей возрастающей последовательности, которая последним элементом содержит x_n. Обозначим эту задачу (и ее решение) за $B(x_1, \ldots, x_n)$. Тогда

$$A(x_1, \ldots, x_n) = \max\{A(x_1, \ldots, x_{n-1}), B(x_1, \ldots, x_n)\}$$
$$B(x_1, \ldots, x_n) = \max_{i: x_i < x_n} \{B(x_1, \ldots, x_i)\} + 1$$

Таким образом, мы свели задачу к меньшим подзадачам, а так же задачам с немного другой формулировкой. Осталось запустить алгоритм рекурсивного решения этой задачи, с запоминанием ответов.

То что мы построили – это динамика вида «для каждого элемента последовательности храним длину наибольшей возрастающей подпоследовательности оканчивающейся в нем». Ее очень удобно вычислять с начала – для одного элемента x_1 ответ очевиден: $B(x_1) = 1$. Для последовательности из k элементов мы идем от x_1 до x_{k-1} и смотрим – если x_k можно подсоединить, и длина больше чем уже была – увеличиваем.

При этом для каждого элемента мы идем по всему префиксу, таким образом алгоритм работает за $O(n^2)$.

Есть другая динамика, которая работает за $O(n \log n)$. Там та же логика – мы пытаемся подсоединить элемент x_n к наибольшей последовательности, но при этом мы дополнительно пользуемся тем, что нам не нужно хранить все последовательности какой-то длины – нам нужно на префиксе хранить лучшую из них (т.е. ту что оканчивается на самое маленькое число). Эта модификация в итоге приводит к тому что поиск подходящей последовательности (наибольшей длины среди тех что можно подсоединить) происходит в упорядоченном массиве, а там работает бинарный поиск.

1.2.3 Расстояние Левенштейна (расстояние редактирования)

Есть два слова, т.е. две последовательности букв $x_1, \ldots, x_n$ и $y_1, \ldots, y_m$. Задача состоит в то чтобы найти наименьшее число изменений (замен/вставок/удалений букв), достаточное чтобы из первого слова сделать второе.

Для ее решения обычно используют двумерную динамику. Для этого предлагается заполнить таблицу d, где $d[i, j]$ – редакторское расстояние для строк $x_1, \ldots, x_i$ и $y_1, \ldots, y_j$ – тогда интересующее нас расстояние находится в ячейке $d[n, m]$. База здесь задается легко, если рассматривать также пустые строки – т.е. ячейки вида $d[i.0]$ и $d[0, j]$. Для них выполнено

$$d[i, 0] = i;\ d[0, j] = j,$$

потому что чтобы получить из слова длины i пустое слово надо сделать ровно i удалений символов (второе аналогично). Теперь таблица выглядит так (слева):

	–	x_1	x_2	x_3	$\cdots$	x_{n-1}	x_n
–	0	1	2	3	$\cdots$	$n-1$	n
y_1	1				$\cdots$		
y_2	2				$\cdots$		
y_3	3				$\cdots$		
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
y_{m-1}	$m-1$				$\cdots$		
y_m	m				$\cdots$		

	–	x_1	x_2	x_3	$\cdots$	x_{n-1}	x_n
–	0	1	2	3	$\cdots$	$n-1$	n
y_1	1				$\cdots$		
y_2	2				$\cdots$		
y_3	3				$\cdots$		
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
y_{m-1}	$m-1$				$\cdots$		
y_m	m				$\cdots$		

Теперь нам надо каким-то образом сделать переход. Посмотрим на клетку $d[1,1]$ (выделена красным справа). Т.е. это кол-во изменений необходимое, чтобы переделать строку (из одного символа) x_1 в строку y_1. Есть два случая:

$$d[1,1] = \begin{cases} 0, & x_1 = y_1 \\ 1, & x_1 \neq y_1 \end{cases}.$$

Теперь немного остановимся и поговорим еще раз о смысле $d[i,j]$. Это число действий, которого хватит чтобы переделать строку $x_1, \dots, x_i$ в $y_1, \dots, y_j$ (будем рассматривать именно в таком направлении, для удобства). Теперь смотрите, если мы за $d[i, j-1]$ переделали строку $x_1, \dots, x_i$ в $y_1, \dots, y_{j-1}$ то дописав один символ мы можем получить $y_1, \dots, y_j$! Это значит, что $d[i,j] \leqslant d[i, j-1] + 1$. Аналогично, $d[i,j] \leqslant d[i-1, j] + 1$.

И, наконец, если $x_i = y_j$ то мы могли превратить только префикс $x_1, \dots, x_{i-1}$ в $y_1, \dots, y_{j-1}$, а x_i оставить тем же, т.е. $d[i,j] \leqslant d[i-1, j-1]$. Или, если они не равны, мы могли превратить префикс и потом еще одним действием заменить x_i на y_j, Итак, все это дает нам вот такую формулу

$$d[i,j] = \min \begin{cases} d[i-1, j-1] + \delta(x_i, y_j) \\ d[i-1, j] + 1 \\ d[i, j-1] + 1 \end{cases} \qquad \text{где } \delta(x_i, y_j) = \begin{cases} 0, & x_i = y_j \\ 1, & x_i \neq y_j \end{cases}$$

А теперь надо понять, корректно ли такое рассуждение. Пока что все что мы получили это какая-то оценка сверху на ответ – но почему он не может быть меньше? Доказывать будем по индукции. Пусть для $d[i-1, j-1], d[i-1, j], d[i, j-1]$ значения были найдены корректно (это действительно минимальные числа изменений). Рассмотрим $d[i,j]$. Есть два случая:

1. $x_i = y_j$, пусть $d[i,j] = t$ (по нашему алгоритму), но на самом деле расстояние между словами меньше и равно p. Заметим, что т.к. $x_i = y_j$ имеем $d[i,j] \leqslant d[i-1, j-1]$, значит $d[i-1, j-1] \geqslant t > p$ – т.е. оно было вычислено некорректно, что противоречит предположению индукции (базу которой мы задали когда вычислили $d[i, 0]$ и $d[0, j]$).

2. $x_i \neq y_j$, тогда, опять же, предположим что есть более быстрый способ переделать строку. Т.к. $x_i \neq y_j$ имеем $d[i-1, j] \geqslant t-1$, $d[i, j-1] \geqslant t-1$ и $d[i-1, j-1] \geqslant t-1$.

   Посмотрим теперь на x_i и y_j в процессе преобразования $x_1, \dots, x_i$ в $y_1, \dots, y_j$ за $d[i,j] = p$ шагов.

   - Пусть x_i была удалена в ходе преобразований. Тогда ее можно было удалить в самом начале – но тогда мы перешли бы к задаче решенной в $d[i-1, j]$, т.е. $d[i,j] = d[i-1, j] + 1 \geqslant t > p$ – противоречит тому что $d[i,j] = p$;
   - Пусть y_j в какой-то момент была добавлена. Опять же, ее можно было бы добавить в конце – и тогда мы аналогично предыдущему пункту переходим к задаче $d[i, j-1]$;
   - Наконец, пусть неверны оба утверждения выше. Но тогда в какой-то момент x_i была заменена на y_j (добавлять и удалять элементы справа мы не могли так как не верно вторoе утверждение), и тогда за одно действие делая такую замену мы переходим в клетку $d[i-1, j-1]$, и опять получаем противоречие.

Теперь, когда мы доказали корректность переходов, надо обсудить сложность. Чтобы получить $d[n, m]$ нам надо заполнить всю таблицу, при этом каждый переход выполняется за $O(1)$, таким образом общая сложность равна $O(nm)$.

Примечание 1.6. Со сложностью по памяти здесь есть интересный момент. Если посмотреть на картинку выше можно понять, что мы можем каждый раз заполнять массив по столбцам, при этом работаем мы только лишь с предыдущим столбцом. Поэтому если мы хотим лишь найти $d[n, m]$ то нам достаточно хранить $O(\min\{n, m\})$ памяти (понятно что так же можно делать и по строкам).

Проблема в том, что в таком случае сложнее будет восстановить саму последовательность изменений, переводящую первую строку во вторую – в случае с таблицей восстанавливать можно идя с конца и смотря откуда был сделан переход – а в случае алгоритма сохраняющего только одну строку мы эту информацию отбрасываем.

Однако есть способ, в котором мы также рассматриваем матрицу $e[i, j]$ – по сути то же самое, но для суффиксов, т.е. у$[i, j]$ это число действий необходимое чтобы превратить $x_i, \dots, x_n$ в $y_j, \dots, y_m$. Она, очевидно, заполняется таким же образом. А дальше применяется метод «разделяй и властвуй», в результате мы получаем и $O(nm)$ по времени, и $O(\min\{n, m\})$ по памяти.

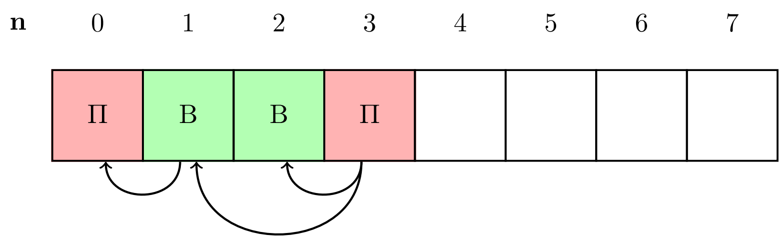
1.2.4 Антагонистические игры

В этих задачах есть два игрока, и некоторый ориентированный граф состояний игры – ребра в этом графе означают что из одного состояния можно перейти в другое. При этом некоторые состояния помечены как проигрышные, некоторые – как выигрышные. И в таких задачах обычно вопрос ставится следующим образом: кто выигрывает если игра начинается в таком-то состоянии, и ходит такой-то игрок?

Рассмотрим пример. Пусть на доске написано число n, и на каждом ходу его можно уменьшить либо на 1, либо на 2. Тот, кто не может сделать ход – проиграл.

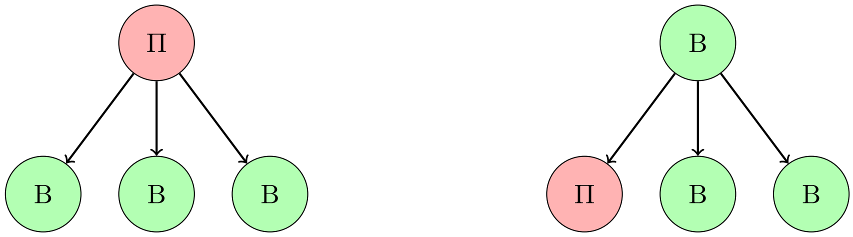
В такой постановке состояния – это числа на доске, а ребра ведут из состояния i в состояния $i - 1$ и $i - 2$. При этом по условию состояние 0 – проигрышное, из него нельзя сделать ход.

Посмотрим на первые несколько состояний. Если на доске написано 1, то мы можем отнять 1 и перейти в 0 – из него ход сделать нельзя. Значит это состояние выигрышное. То же и с 2. Если же на доске написано 3, то возникает проблема – мы можем перейти только в состояния 1 и 2 – а они выигрышные для того кто в них начинает. Таким образом 3 – проигрышное. Итак, выглядит пока что это так:



Собственно здесь уже видна основная идея. Когда мы рассматривали состояние 3 мы поняли что куда бы мы из него не перешли мы попадем в состояние из которого следующий игрок сможет победить. Значит для нас состояние в котором мы находимся – проигрышное.

Но если есть хотя бы одно проигрышное для второго игрока состояние (т.е. состояние в которое мы можем перейти из нашего, и которое является проигрышным для того кто из него начинает) – то почему туда не перейти? В итоге мы приходим к следующим двум общим правилам:



Т.е. если из данного состояния мы можем перейти **только в выигрышные** – оно проигрышное. И если есть **хотя бы одно** проигрышное состояние в которое мы можем перейти из нашего – оно выигрышное.

И этот принцип работает для всех задач которые можно задать как задачу выше – набор состояний, переходы и некоторые выделенные конечные состояния. Разматывая от них мы для любого состояния можем найти, является ли оно выигрышным, или нет.

1.2.5 Задача о рюкзаке

Наконец обсудим задачу о рюкзаке. Формулировок у нее довольно много, рассмотрим одну из них:

Задача. Дано n предметов, W — вместимость рюкзака, и для каждого из n объектов есть пара (c_i, w_i) – его стоимость и вес. Найти подмножество этих объектов такое что его суммарный вес не превосходит W, а суммарная стоимость – максимальна.

Иначе говоря, рассмотрим набор бинарных величин $B = \{b_1, \dots, b_n\}$, $b_i \in \{0, 1\}$ (смысл следующий: $b_i = 1$ если i-й объект мы хотим взять). Тогда задача будет иметь такой вид:

$$\sum_{i=1}^{n} b_i c_i \to \min_{B} \text{ при условии } \sum_{i=1}^{n} b_i w_i \leqslant W.$$

Можно, конечно, решать эту задачу перебором всех подмножеств, но это долго. Поэтому предлагается рассмотреть следующую динамику: пусть $A[i, j]$ – ответ для этой задачи если разрешено использовать только первых j объектов, и вес рюкзака не превышает i. Тогда мы опять как в пункте про расстояние редактирования получаем таблицу (слева):

	1	2	3	$\cdots$	$n-1$	n
0	0	0	0	$\cdots$	0	0
1				$\cdots$		
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
$w_1 - 1$				$\cdots$		
w_1				$\cdots$		
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
$W - 1$				$\cdots$		
W				$\cdots$		

	1	2	3	$\cdots$	$n-1$	n
0	0	0	0	$\cdots$	0	0
1	0			$\cdots$		
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
$w_1 - 1$	0			$\cdots$		
w_1	c_1			$\cdots$		
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
$W - 1$	c_1			$\cdots$		
W	c_1			$\cdots$		

При этом верхний ряд заполнен нулями, т.к. в рюкзак веса 0 (вес отсчитывается по вертикали) мы не можем ничего положить. Теперь, будем заполнять эту таблицу по столбцам. Заметим, что в первом столбце начиная с какого-то момента (а именно когда вес рюкзака становится больше w_1) будет лежать c_1 – это единственный объект который мы можем взять.

Теперь рассмотрим заполнение k-го столбца. Рассмотрим $d[i, k]$. Если $i < w_k$ то мы просто не можем взять k-й объект – значит ответ это ответ для множества из первых $k-1$ объекта, т.е. $d[i, k-1]$. Иначе мы можем попробовать взять в рюкзак k-й объект – но тогда у нас останется всего $i - w_k$ веса, и в оставшееся нам надо набрать по максимуму – а решение для $d[i - w_k, k-1]$ у нас уже посчитано на прошлом шаге. Поэтому формула перехода выглядит так:

$$d[i, j] = \max \begin{cases} d[i, j-1], \\ d[i - w_j, j-1] + c_i, \quad i > w_j, \end{cases}$$

и в таблице (зеленым отмечено то что мы посчитали ранее, на предыдущих шагах):

	1	2	3	$\cdots$	j-1	j	$\cdots$	$n-1$	n
0	0	0	0	$\cdots$	0	0	$\cdots$	0	0
1				$\cdots$			$\cdots$		
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
$w_j - 1$				$\cdots$			$\cdots$		
w_j				$\cdots$			$\cdots$		
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
$W-1$				$\cdots$			$\cdots$		
W				$\cdots$			$\cdots$		

красным показан первый элемент в столбце j для которого ответ формируется на основе двух вариантов – либо $c_j + d[0, j-1]$, либо просто $d[i, j-1]$. Для предыдущих нельзя взять j-й объект, поэтому вариант всего один (показан красными стрелками).

Корректность здесь как раз следует из построения – если мы не берем j-й объект то наш ответ это $d[i, j-1]$. А если берем, то из монотонности по весу (в рюкзак большего веса умещается не меньше по стоимости) получаем что надо брать ответ именно из $d[i - w_j, j-1]$. Сложность – опять же размер таблицы, $O(nW)$. И если нам нужно только **посчитать** максимальную стоимость то мы можем поддерживать лишь предпоследний слой, таким образом по памяти сложность может быть уменьшена до $O(W)$.