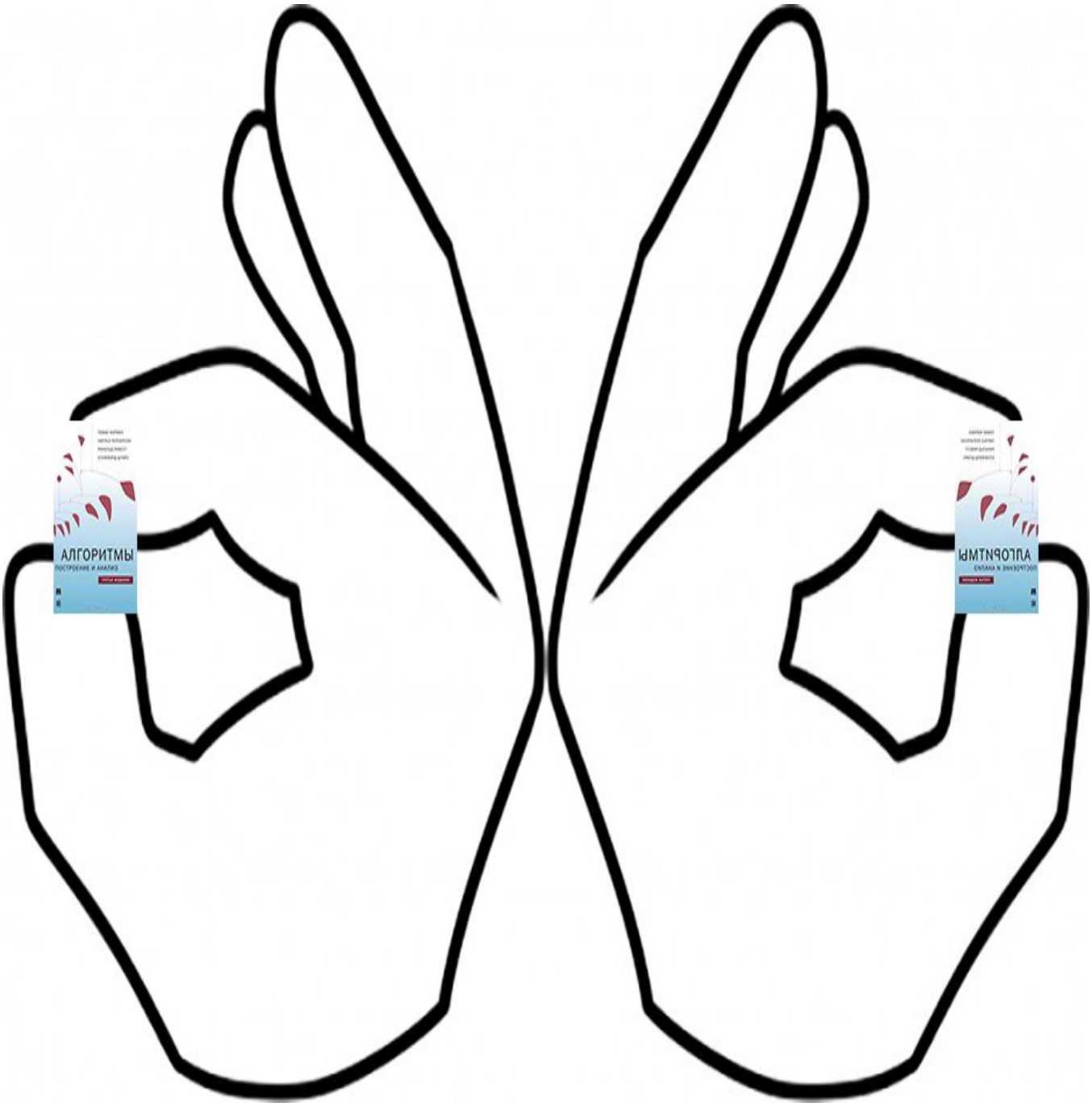
Алгоритмы и Структуры Данных

Мини-Кормен 2



Вадим Гринберг

по лекциям Сергея Александровича Объедкова

Содержание

Машины Тьюринга

Детерминированные МТ

Определение 1. ***Машина Тьюринга (МТ)*** *– это кортеж из семи элементов:* $\langle Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject}\rangle$*, где*

- Q *– это множество состояний машины*
- Σ *– это алфавит*
- Γ *– это рабочий алфавит, в котором помимо алфавита* Σ *есть также различные вспомогательные символы, такие как пробел* $\_$*, символ-разделитель* $\#$ *и другие.*
- δ *– функция перехода:* $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\mathcal{L}, \mathcal{R}\}$*, которая сопоставляет паре из состояния и символа, на который сейчас смотрит головка МТ, тройку из нового состояния, символа, который головка записала на ленту, и направления, куда сдвинулась головка – налево (*$\mathcal{L}$*), или направо (*$\mathcal{R}$*).*

  Пример: $(q_{25}, 'b') \to (q_7, 'a', \mathcal{L})$ *– изначально МТ была в состоянии 25, и головка смотрела на символ* b*. Затем МТ перешла в состояние 7, записала в ячейку символа* b *на ленте символ* a*, и сдвинулась на ячейку влево.*
- q_0 *– начальное состояние МТ*
- q_{accept} *– допущение входной строки, состояние остановки*
- q_{reject} *– отвержение входной строки, состояние остановки*

Это было определение одноленточной МТ. Определение многоленточной Машины Тьюринга отличается функцией перехода δ, которая для МТ с k лентами примет следующий вид: $\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{\mathcal{L}, \mathcal{R}\}^k$, то есть, учитываются символы, на которые смотрит головка каждой из k лент, и движения этих k головок.

Замечание: Можно также добавить символ движения $\mathcal{S}$, соответствующий случаю, когда головка не перемещается, а остаётся на месте, но обычно в этом нет необходимости.

Замечание: $q_{accept} \neq q_{reject}$. При этом q_0 либо одно из данных состояний, либо не совпадает с ними.

Определение 2. ***Разрешаемый язык****, или* ***Множество допускаемых слов*** $\mathscr{L}(M) = \{w \in \Sigma^* \mid$ *машина* M *допускает слово* $w\}$*. Иначе говоря, это множество слов, таких, что, получив на вход слово из данного множества, Машина Тьюринга* M *в результате работы придёт в состояние* q_{accept}*.*

В частности, если $q_0 = q_{accept}$, то $\mathscr{L}(M) = \{\Sigma^*\}$, а если $q_0 = q_{reject}$, то $\mathscr{L}(M) = \varnothing$.

Утверждение 1. *Одноленточные и многоленточные Машины Тьюринга эквивалентны в смысле полиномиальности времени работы.*

Доказательство.

- 1-МТ $\to$ k-МТ

  Заметим, что одноленточная МТ – частный случай многоленточной, поэтому в эту сторону и доказывать нечего.
- k-МТ $\to$ 1-МТ

  Будем строить одноленточную Машину Тьюринга, эквивалентную имеющейся k-ленточной МТ. Предпримем следующие действия:

1. Добавим в рабочий язык Γ символ-разделитель #, которым мы разделим ленту по порядку на k частей, каждая из которых будет соответствовать определённой строке k-ленточной МТ.
2. Нужно при этом помнить, где находится (на какой смотрит символ) каждая из k головок многоленточной МТ. Для этого на нашей одноленточной МТ в каждой из частей, соответствующих своим лентам в k-МТ, будем помечать символы (фактически добавляя в алфавит такой же символ, но "помеченный"), на которые смотрит головка МТ на соответственно каждой ленте.
3. Начальное состояние МТ будет выглядеть так:

$$\overset{\nabla}{c}w\underbrace{\#, \#, \ldots, \#}_{k-1}$$

где c – первый символ входного слова, w – оставшаяся его часть. Головка смотрит на c. Между разделителями ничего пока что нет (так как в k-МТ входное слово изначально появляется только в первой строке).

Теперь проанализируем время имитации одноленточной МТ одного шага k-ленточной Машины Тьюринга. Посчитаем для начала время одного шага k-МТ. Ясно, что T_k – время одного шага = O(число всех занятых ячеек). Однако, на каждой ленте может находиться не более $t(n)$ символов, где t – какая-то полиномиальная функция, характеризующая время работы многоленточной МТ на каком-то входе размера n. Всего лент k, значит, на всех лентах суммарно не более $k \cdot t(n)$ занятых ячеек. Отсюда получаем, что размер ленты нашей одноленточной Машины Тьюринга составляет $O\big(t(n)\big)$. Тогда для того, чтобы сымитировать один шаг k-ленточной МТ, нам понадобится $O\big(t(n)^2\big)$ времени, что полиномиально зависит от времени одного шага на многоленточной МТ.

Итак, мы доказали, что одноленточные и многоленточные Машины Тьюринга эквивалентны в смысле полиномиальности времени работы. □

Недетерминированные МТ

Определение 3. ***Недетерминированная Машина Тьюринга (НМТ)*** *– та же обычная МТ, которая отличается только функцией перехода δ:*

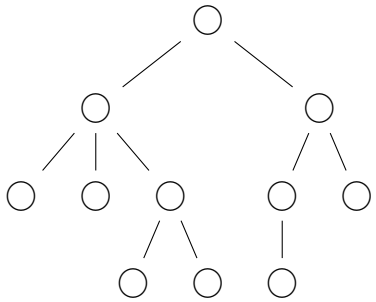
$$\delta : Q \times \Gamma \to 2^{Q \times \Gamma \times \{\mathcal{L},\, \mathcal{R}\}}$$

которая сопоставляет паре из состояния и символа, на который сейчас смотрит головка МТ, подмножество из троек типа (состояние, символ, движение).

Пример: $(q_{25}, 'b') \to \begin{cases} (q_3, 'a', \mathcal{L}) \\ (q_4, 'c', \mathcal{R}) \end{cases}$ – выполняется лишь один из переходов.

Машины Тьюринга можно представлять в виде деревьев выбора с ветвями. Тогда ДМТ выглядит так (◯ обозначает состояние):

А НМТ, соответственно, выглядит так:



Определение 4. *Говорят, что Машина Тьюринга* ***допускает*** *слово w, если в её дереве выбора, соответствующем данному входному слову, есть хотя бы один лист, отвечающий состоянию q_{accept}.*

Можно провести аналог с ДКА и НКА. В частности, ДМТ допускает слово, если приходит в состояние q_{accept}, а НМТ допускает слово, если в q_{accept} приходит хотя бы одна из её ветвей.

Замечание: **Временем работы** МТ является максимальная длина ветви дерева данной Машины Тьюринга.

Основы теории вычислительной сложности. Основные классы

Абстрактные задачи. Кодирование.

Ранее нами в курсе алгоритмов разбирались задачи, для которых есть алгоритм с полиномиальной сложностью. Это означает, что если размер входа равен n, то сложность этого алгоритма равна $O(n^k)$ для какой-то константы k. Можно ли найти такой алгоритм для *любой* задачи? К сожалению, нет. Стоит вспомнить знаменитую "проблему останова", которую невозможно решить на компьютере, сколько бы времени на неё не выделяли. Также есть примеры задач, для которых есть алгоритмы решения, но их сложность не является полиномиальной. Поэтому часто говорят, что полиномиальные задачи "просты".

Перед тем, как идти дальше, нужно понять — а как формально ввести понятие "задача"? Введём его так:

Определение 5. ***Задача*** Q *— это бинарное отношение на множестве* I *(множестве* ***экземпляров****) и множестве* S *(множестве* ***решений****).*

В теории вычислительной сложности в основном рассматривают **задачи разрешимости**, т.е. задачи, в которых $S = \{0, 1\}$ — "нет" или же "да".

Для того, чтобы компьютер решил задачу, нам необходимо передавать ему входные данные в понятном для него виде. Для этого используется **кодировка** — представление входных данных в виде слов над бинарным алфавитом. Вообще, всё может быть закодировано:

- **Кодирование целого числа** – полиномиально связано с его двоичным представлением.
- **Кодирование конечного множества** – полиномиально связано с его кодированием в виде списка элементов.
- **Кодирование графа** – полиномиально связано с количеством узлов в нём. Можно кодировать как в виде списка узлов и рёбер, так и в виде матрицы смежности.

Класс P

Будем полагать, что множество экземпляров — это $\{0, 1\}^*$. Тогда язык, соответствующий задаче Q можно определить так:

$$L = \{x \in \{0, 1\}^* \mid Q(x) = 1\}$$

Теперь введём понятие **класса задач** P (Polynomial):

Определение 6.

$$\mathsf{P} = \{\mathscr{L}(M) \mid M \text{ — } MT \text{ с полиномиальным временем работы}\}$$

Другими словами: задача лежит в классе P тогда и только тогда, когда для соответствующего ей языка можно построить допускающую детерминированную МТ с полиномиальным временем работы.

Теперь рассмотрим пару задач:

- Задача о поиске пути между вершинами в графе. Очевидно, что она лежит в P, так как входные данные представляют собой вершины и рёбра, а алгоритм решения (обход в ширину) полиномиален от количества вершин и рёбер.

- Задача проверки чисел на взаимную простоту: на вход подаются два числа x и y, каждое из которых кодируется n символами. Взаимно просты ли они? Попробуем составить наивный алгоритм проверки:

Алгоритм 1 Наивная проверка чисел на взаимную простоту.

Ввод: Пара чисел x, y
Вывод: Ответ на вопрос "Верно ли, что x и y взаимно просты?"

```
1: for t = 1 to min(x, y) do
2:     if t | x and t | y then
3:         return false
4: return true
```

Как видно, он требует $O(\min(x, y))$ операций. Так как каждое число кодируется с помощью n нулей и единиц, то итоговая сложность составляет $O(2^n)$. Не подходит.

Значит ли это, что эта задача не имеет полиномиального решения? *Нет.* Рассмотрим модификацию этого алгоритма, использующую алгоритм Евклида:

Алгоритм 2 Проверка чисел на взаимную простоту с использованием алгоритма Евклида.

Ввод: Пара чисел x, y
Вывод: Ответ на вопрос "Верно ли, что x и y взаимно просты?"

```
1: r ← x mod y
2: while r ≠ 0 do
3:     x ← y
4:     y ← r
5:     r ← x mod y
6: return y = 1
```

Теперь заметим следующее: за каждые две итерации цикла одно из чисел уменьшается по крайней мере вдвое. Для этого достаточно рассмотреть два случая — когда одно число меньше удвоенного второго или же больше его. Тогда алгоритм будет требовать $O(\log(\min(x, y))$ операций. Следовательно, сложность алгоритма составляет $O(n)$ и эта задача принадлежит классу P.

Ранее мы говорили, что для того, чтобы компьютер решил задачу, входные данные нужно закодировать. Может ли кодировка повлиять на время работы алгоритма? Рассмотрим следующую задачу:

Вход: $k \in \mathbb{Z}$

Время работы: $O(k)$

Теперь рассмотрим, как будет зависить время работы от кодировки:

- **Унарная кодировка**: $n = k \implies O(n)$
- **Бинарная кодировка**: $n = \lfloor \log_2 k \rfloor + 1 \implies O(2^n)$

Важно выбирать правильную кодировку.

Стоит сказать, что, например, двоичная и десятичная кодировки полиномиально связаны (т.е. время работы алгоритма в одном и в другом случае будет отличаться с точностью до умножения на полином какой-то степени).

Классы NP и coNP

В теории вычислительной сложности также существует класс NP, который прославился с помощью гипотезы P = NP. Введём формальное определение:

Определение 7. ***Класс*** NP *— это класс задач таких, что они могут быть* ***верифицированы*** *за полиномиальное время.*

Что значит "верифицировать" задачу? Если язык $\mathscr{L}$ лежит в классе NP, то существует функция от двух аргументов $A(x, y)$ — **алгоритм верификации** — с полиномиальной сложностью от x такая, что x лежит в $\mathscr{L}$ тогда и только тогда, когда для него существует y (его принято называть **сертификатом**) такой, что $A(x, y) = 1$. При этом сертификат должен быть полиномиально зависим от размера x.

Достаточно логично, что $\mathsf{P} \subseteq \mathsf{NP}$, так как для задачи из класса P алгоритмом верификации служит сам алгоритм решения.

Пример задачи из класса NP — задача о распознании составных чисел.

Возникает вопрос: а есть ли задача, которую нельзя верифицировать за полиномиальное время? *Да.* Приведём пример, предварительно введя определение:

Определение 8. *Неориентированный граф называется* ***Гамильтоновым****, если он содержит* ***Гамильтонов цикл*** *– простой цикл, содержащий все вершины графа.*

Алгоритм 3 Проверка графа на Гамильтоновость.

Ввод: граф $G = (V, E)$

Вывод: Ответ на вопрос: "Содержит ли граф G гамильтонов цикл?"

сформировать список перестановок вершин графа G

проверить каждую перестановку

Проанализируем время работы. Пусть граф G дан на вход в виде матрицы смежности размера n. Тогда $|V| = O(\sqrt{n})$, и время работы всего алгоритма проверки каждой перестановки составит $O(\sqrt{n!})$.

Ещё один пример задачи, которая не факт, что входит в NP (на самом деле, она из coNP, о котором будет сказано позднее):

Ввод: граф G

Вывод: Ответ на вопрос: "Правда ли, что граф G не гамильтонов?"

В данном случае сертификат будет экспоненциален от размера входных данных, так как придётся проверить все циклы в графе, которых экспоненциальное от числа вершин количество.

Ранее было введено определение недетерминированной машины Тьюринга. Теперь можно сформулировать другое определение класса NP (Nondeterministic Polynomial, а не Not Polynomial, как может сначала показаться):

Определение 9. ***Класс*** NP *— это множество языков $\mathscr{L}(M)$, где M — это недетерминированная машина Тьюринга с полиномиальным временем работы.*

Возникает разумный вопрос — а почему это определение равносильно предыдущему? Ясно, что по алгоритму верификации можно построить недетерминированную Машину Тьюринга (случайно выбираем сертификат и запускаем работу). Обратно: если имеется НМТ, то путь по ветвям, оканчивающийся q_{accept}, и является сертификатом.

Также принято параллельно с классом NP вводить класс coNP:

Определение 10.

$$\mathsf{coNP} = \left\{\overline{\mathscr{L}} = \Sigma^* \setminus \mathscr{L} \mid \mathscr{L} \in \mathsf{NP}\right\}$$

Иначе говоря, класс coNP суть класс дополнений языков из NP. В качестве упражнения покажите, что $\mathsf{P} \subseteq \mathsf{coNP}$.

NP-полные задачи

Замечание: По поводу терминологии: вообще понятия "языка"и "задачи"в теории алгоритмов являются различными, но мы их используем почти как синонимы. Формально, задача – это «условие» и «ответ» (**да** или **нет**). А язык – это множество условий, на которые ответ **да**. Просто, когда мы говорим о языке, мы рассматриваем МТ, разрешающую (допускающую) данный язык, то есть, определяющую, лежит ли поданное на вход слово в рассматриваемом языке. В то же время, если говорить о задаче, то мы рассматриваем МТ, решающую данную задачу (на которую ответом будет **да** или **нет**, что аналогично допущению языка). Поэтому можно говорить о классах P и NP как о классах языков, так и о классах задач.

Введём пока что условное определение NP-полных задач.

Определение (неформальное)**.** ***Класс*** NPC *— это множество самых трудных задач, которые далеко не факт, что имеют решение за полиномиальное время.*

Один из самых важных вопросов современной науки об алгоритмах – это вопрос "совпадают ли классы P и NP"(P = NP ??). Причём оба возможных варианта ответа – **да** или **нет**. Ответ неизвестен, однако понятие NP-полноты, облегчает поиск ответа на данный вопрос, **каким бы ответ ни был**.

- **ДА** – тогда достаточно взять NPC задачу и придумать к ней полиномиальное решение.
- **НЕТ** – привести пример такой NP-полной задачи NP, которая точно не лежит в P.

Проблема в том, что неизвестно, есть ли что-то помимо NPC и P, так как некоторые NP задачи на самом деле являются P. Более того, вполне может оказаться, что даже для NPC задачи существует полиномиальное решение. Отсюда разделение на подобные классы весьма условно.

Будем рассматривать некоторые задачи NPC.

Алгоритм 4 Задача *SAT*

Ввод: Булева формула
Вывод: Ответ на вопрос: “выполнима ли булева формула??”

Определение 11. *Булева формула называется* ***выполнимой****, если существует такое* ***означивание*** *– упорядоченная комбинация значений переменных формулы – что формула при данном означивании принимает значение 1 (Истина).*

Пример выполнимой булевой формулы: $x \vee y$. Пример невыполнимой булевой формулы: $x \wedge \neg x$. Одним из возможных решений задачи SAT является полный перебор всех возможных означиваний, однако такое решение будет экспоненциально зависеть от числа переменных.

Несложно заметить, что $SAT \in \mathsf{NP}$, так как существует полиномиальный алгоритм верификации, сертификатом для которого будут просто значения переменных при означивании. Хороший вопрос, решается ли задача SAT за полиномиальное время. Была сформулирована и доказана следующая теорема:

Теорема Кука-Левина.

$$SAT \in \mathsf{P} \Longleftrightarrow \mathsf{P} = \mathsf{NP}$$

В ней, по факту, подразумевается следующее: если для задачи SAT существует решение за полиномиальное время, то для всех остальных языков (задач) класса NP также существует решение за полиномиальное время. Как это вообще возможно? Суть в том, что все эти языки **сводятся** к задаче SAT за полиномиальное время. Дадим строгое определение **сводимости**.

Определение 12. *Функция $f : \Sigma^* \to \Sigma^*$ называется* ***вычислимой за полиномиальное время****, если существует такая ДМТ с полиномиальным временем работы, которая, получая на вход слово $w \in \Sigma^*$, заканчивает работу с $f(w)$ на ленте.*

Определение 13. *Язык A* ***сводится*** *к языку B за полиномиальное время, если существует функция f, вычислимая за полиномиальное время, такая, что $w \in A \iff f(w) \in B$.*

Обозначение: $A \leqslant_p B$.

Докажем основное утверждение о сводимости:

Утверждение 2. *Пусть $A \leqslant_p B$ и $B \in \mathsf{P}$ (существует полиномиальный алгоритм решения задачи). Тогда $A \in \mathsf{P}$.*

Доказательство. Достаточно построить алгоритм, показывающий, что для языка A тоже существует полиномиальное решение. Пусть у нас имеются Машины Тьюринга M_A, допускающая язык A, и M_B, допускающая язык B. Поскольку $A \leqslant_p B$, то $\exists$ полиномиальная f, удовлетворяющая определению выше. Построим алгоритм вычисления M_A:

Ввод: слово w
вычислить $f(w)$
return $M_B\big(f(w)\big)$

Поскольку сама функция f является вычислимой за полиномиальное время, то вычисление $f(w)$ потратит полиномиальное время. Кроме того, поскольку $B \in \mathsf{P}$, то вычисление $M_B\big(f(w)\big)$ также займёт полиномиальное время (от размера $f(w)$, но $f(w)$ также вычисляется за полином)! Отсюда $M_A(w)$ также допускает слово w за полиномальное время, что по определению означает, что $A \in \mathsf{P}$. □

Теперь можно ввести два важных определения:

Определение 14. *Язык $\mathscr{L}$ – NP-трудный, если для любого языка $\mathscr{L}' \in \mathsf{NP}$ выполнено, что $\mathscr{L}' \leqslant_p \mathscr{L}$. (Сам $\mathscr{L}$ не обязательно лежит в NP).*

Определение 15. *Язык $\mathscr{L}$ – NP-полный, если $\mathscr{L} \in \mathsf{NP}$ и $\mathscr{L}$ – NP-трудный.*

Примеры полиномиальных сведений

Рассмотрим следующую задачу, именуемую **Задачей о КНФ**, или же $3SAT$:

Алгоритм 5 Задача $3SAT$
Ввод: 3-КНФ φ – конъюнкция дизъюнкций из 3-х элементов
Вывод: Ответ на вопрос: "выполнима ли φ??"

Пример 3-КНФ: $(\neg x \vee y \vee \neg z) \wedge (x \vee \neg y \vee z)$. Она выполнима – решением будет означивание из всех 0. В более же сложных случаях весьма непросто ответить на поставленный вопрос.

Однако, данную задачу можно свести к другой, именуемой $NOT-ALL-EQUAL-3SAT$ $(NAE-3SAT)$:

Алгоритм 6 Задача $NAE-3SAT$
Ввод: 3-КНФ φ
Вывод: Ответ на вопрос: "существует ли такое выполняющее означивание, что ни в какой скобке значения всех 3-х литер не одинаковы??"

Сведём задачу $3SAT$ к $NAE-3SAT$. Но делать это будем не сразу, а провернём сначала один приём: сведём задачу $3SAT$ к так называемой $NAE-4SAT$ – задаче с теми же условиями, что и для $NAE-3SAT$, только с 4 литерами в скобках. Для этого возьмём функцию f, вычислимую за полиномиальное время, которая получает на вход 3-КНФ φ, и каждую скобку переписывает так:

1. Пусть есть дизъюнкция $(l_1 \vee l_2 \vee l_3)$
2. Заменим её на скобку вида $(l_1 \circ l_2 \circ l_3 \circ t)$, добавив переменную $t \notin \varphi$, одну и ту же для всех скобок. $\circ$ условно обозначает следующую операцию: *выражение из элементов, связанных $\circ$, истинно тогда и только тогда, когда среди связанных литер не все литеры одинаковы.*

Наша форма из примера выше приобретёт следующий вид: $(\neg x \circ y \circ \neg z \circ t) \wedge (x \circ \neg y \circ z \circ t)$.

Утверждение 3. $3SAT \leqslant_p NAE-4SAT$

Доказательство. Докажем по определению, используя введённую ранее функцию f: $\varphi \in 3SAT \iff f(\varphi) \in NAE-4SAT$. Иными словами, нужно показать, что φ является выполнимой $3SAT$ формулой тогда и только тогда, когда $f(\varphi)$ является выполнимой $NAE-4SAT$ формулой.

- $\varphi \in 3SAT \implies f(\varphi) \in NAE-4SAT$

  Пусть $\varphi \in 3SAT$. Тогда существует такое означивание, что φ истинно. Зафиксируем это выполняющее означивание. При данном означивании в каждой скобке $(l_1 \circ l_2 \circ l_3 \circ t)$ хотя бы одна из литер l_1, l_2, l_3 истинна. По свойству NAE-КНФ, чтобы скобка была истинна, не все литеры должны быть одинаковы. Но тогда сделаем t ложной (равной 0) – таким образом мы добьёмся выполнения условий задачи $NAE-4SAT$. В итоге мы, применив функцию f к $\varphi \in 3SAT$, получили $f(\varphi) \in NAE-4SAT$.

- $f(\varphi) \in NAE-4SAT \implies \varphi \in 3SAT$

  Пусть $f(\varphi)$ – выполнимая $NAE-4SAT$ КНФ. Рассмотрим выполняющее означивание для данной формы, перебрав все возможные варианты переменной t:

– $t = 0$. В таком случае в каждой скобке $(l_1 \circ l_2 \circ l_3 \circ t)$ хотя бы одна литера истинна (аналогично рассуждению выше). Но тогда, соотвественно, $(l_1 \vee l_2 \vee l_3) = 1$, и рассматриваемое означивание является выполняющим и для φ.

– $t = 1$. В таком случае в каждой скобке $(l_1 \circ l_2 \circ l_3 \circ t)$ хотя бы одна литера ложна (аналогично рассуждению выше). Но тогда можно просто инвертировать все литеры – тогда t станет равной 0, в каждой скобке хотя бы одна литера истинна, что приведёт нас к предыдущему случаю. Но тогда означивание, обратное рассматриваемому, будет выполняющим для φ.

В обоих случаях мы доказали существование выполняющего означивания для $\varphi \in 3SAT$. Отсюда напрямую следует, что $f(\varphi) \in NAE-4SAT \Longrightarrow \varphi \in 3SAT$.

Таким образом, наше утверждение доказано в обе стороны, и $3SAT \leqslant_p NAE-4SAT$. □

Теперь приступим к доказательству следующей сводимости:

Утверждение 4. $NAE-4SAT \leqslant_p NAE-3SAT$

Доказательство. Зададим следующую функцию f: пусть у нас имеется скобка $(l_1 \circ l_2 \circ l_3 \circ l_4) = c_i$ – один из конъюнктов (i-й) типа $NAE-4SAT$. Наша функция f будет сопоставлять скобке c_i конъюнкцию из пары скобок: $f(c_i) = (l_1 \circ l_2 \circ \xi_i) \wedge (l_3 \circ l_4 \circ \neg\xi_i)$, где ξ_i – новая переменная, уникальная для каждой скобки. Докажем по определению, используя данную функцию f, что $\varphi \in NAE-4SAT \Longleftrightarrow f(\varphi) \in NAE-3SAT$.

- $\varphi \in NAE-4SAT \Longrightarrow f(\varphi) \in NAE-3SAT$

  Пусть $\varphi \in NAE-4SAT$ выполнима. Тогда рассмотрим выполняющее означивание. В данном означивании некая скобка $c_i = (l_1 \circ l_2 \circ l_3 \circ l_4)$ принимает значение 1. По свойству $NAE-4SAT$, в скобке должна содержаться хотя бы одна истина – какая-то l_k, $k \in \{1, 2, 3, 4\}$, и хотя бы одна ложь – какая-то l_r, $r \in \{1, 2, 3, 4\}$, причём $k \neq r$. Переберём все возможные варианты k и r:

  – $[k, r] = [1, 2]$, что аналогично варианту $[k, r] = [3, 4]$. Рассмотрим первый. В данном случае в выражении $(l_1 \circ l_2 \circ \xi_i) \wedge (l_3 \circ l_4 \circ \neg\xi_i)$ первая скобка истинна вне зависимости от значения ξ_i. Тогда подбираем значениие ξ_i таким, чтобы $l_3 \circ l_4 \circ \neg\xi_i = 1$ и выполнялись свойства $NAE-3SAT$ – тогда выбранное означивание для $NAE-4SAT$ станет выполняющим и для $NAE-3SAT$.

  – $k = [1, 2]$, $r = [3, 4]$. В данном случае делаем ξ_i ложным (равным 0). В таком случае в скобке $(l_1 \circ l_2 \circ \xi_i)$ есть как истина (l_k), так и ложь (ξ_i), и в скобке $(l_3 \circ l_4 \circ \neg\xi_i)$ есть как истина $(\neg\xi_i)$, так и ложь (l_r). Тогда каждая из скобок будет принимать значение 1, отвечая свойствам $NAE-3SAT$, и выбранное означивание для $NAE-4SAT$ будет выполняющим и для $NAE-3SAT$.

  Отсюда следует, что получившееся $f(\varphi) \in NAE-3SAT$, и, соотвественно, $\varphi \in NAE-4SAT \Longrightarrow f(\varphi) \in NAE-3SAT$.

- $f(\varphi) \in NAE-3SAT \Longrightarrow \varphi \in NAE-4SAT$

  Пусть $f(\varphi)$ выполнима. Тогда рассмотрим выполняющее означивание. Пускай в полученной $f(\varphi)$ есть конъюнкция скобок $(l_1 \circ l_2 \circ \xi_i) \wedge (l_3 \circ l_4 \circ \neg\xi_i)$.

  Предположим, что $l_1 = l_2 = l_3 = l_4 = 1$. Но тогда, чтобы удовлетворять свойствам $NAE-3SAT$, в скобке $(l_1 \circ l_2 \circ \xi_i)$ ξ_i должно быть равно 0, и в то же время в скобке $(l_3 \circ l_4 \circ \neg\xi_i)$ $\neg\xi_i$ также должно быть равно 0, что невозможно. Аналогично для случая

$l_1 = l_2 = l_3 = l_4 = 0$. Значит, случай, когда все литеры l_1, l_2, l_3, l_4 равны, получиться не мог, и хотя бы одна литера должны отличаться от остальных. Но это означает, что в исходной скобке $(l_1 \circ l_2 \circ l_3 \circ l_4)$ не все литеры равны между собой, и скобка удовлетворяет свойствам $NAE-4SAT$. Отсюда уже следует, что если полученная форма выполнима в смысле $NAE-3SAT$, то исходная форма также выполнима в смысле $NAE-4SAT$. А это, собственно, ровно то, что надо было доказать: $f(\varphi) \in NAE-3SAT \Longrightarrow \varphi \in NAE-4SAT$.

Таким образом, наше утверждение доказано в обе стороны, и $NAE-4SAT \leqslant_p NAE-3SAT$. □

Теперь можно легко доказать ранее высказанное утверждение:

Утверждение 5. $3SAT \leqslant_p NAE-3SAT$

Доказательство.

$$\begin{cases} 3SAT \leqslant_p NAE-4SAT \\ NAE-4SAT \leqslant_p NAE-3SAT \end{cases} \Longrightarrow 3SAT \leqslant_p NAE-3SAT$$

□

NPC-языки

Повторим определения, введённые на прошлой лекции:

Определение 16. *Язык* $\mathscr{L}$ – NP-*трудный, если* $\forall\ \mathscr{L}' \in \mathsf{NP} : \mathscr{L}' \leqslant_p \mathscr{L}$. *(Сам* $\mathscr{L}$ *не обязательно лежит в* NP*).*

Определение 17. *Язык* $\mathscr{L}$ – NP-*полный, если:*

1. $\mathscr{L} \in \mathsf{NP}$
2. $\mathscr{L}$ – NP-*трудный*

NP-*полные языки обозначаются как* NPC *(Nondetermenistic Polynomial Complete).*

Докажем пару теорем.

Теорема 1. *Пусть* $\mathscr{L} \in \mathsf{NPC}$. *Тогда, если* $\mathscr{L} \in \mathsf{P}$, *то* $\mathsf{P} = \mathsf{NP}$.

Доказательство. Пусть $\mathscr{L}' \in \mathsf{NP}$. Так как $\mathscr{L} \in \mathsf{NPC}$, то $\mathscr{L}' \leqslant_p \mathscr{L}$. Но, поскольку $\mathscr{L} \in \mathsf{P}$, то $\mathscr{L}' \in \mathsf{P}$, откуда $\mathsf{P} = \mathsf{NP}$.

□

Теорема 2. *Если* $B \in \mathsf{NPC}$ *и* $B \leqslant_p C \in \mathsf{NP}$, *то* $C \in \mathsf{NPC}$.

Доказательство. Проверим оба условия, входящих в определение NP-полного класса.

1. $C \in \mathsf{NP}$ – по условию.
2. Пусть $\mathscr{L}' \in \mathsf{NP}$. Так как $B \in \mathsf{NPC}$, то $\mathscr{L}' \leqslant_p B$, что по определению означает, что существует некая функция $f_{\mathscr{L}'B}$, вычислимая за полиномиальное время. Кроме того, $B \leqslant_p C$, что по определению означает, что существует некая функция f_{BC}, вычислимая за полиномиальное время.

   Но это означает, что $\mathscr{L}' \leqslant_p C$, так как существует функция $f_{\mathscr{L}'C}$, такая, что $f_{\mathscr{L}'C}(w) = f_{BC}\big(f_{\mathscr{L}'B}(w)\big)$ – она также является вычислимой за полиномиальное время.

   Следовательно, $\forall\ \mathscr{L}' \in \mathsf{NP} : \mathscr{L}' \leqslant_p C$, откуда C – NP-трудный по определению.

Оба условия выполняются, значит, $C \in \mathsf{NPC}$. □

Посмотрим более пристально на задачу о выполнимости булевой формулы. **Задача** SAT
Ввод: Булева формула $\varphi(\vec{x})$, где $\vec{x}$ – булев вектор $(x_1, x_2, \ldots, x_n)$
Вывод: Ответ на вопрос: существует ли $\vec{x}$ такой, что $\varphi(\vec{x}) = 1$?

Задачи различных классов можно трактовать несколько иначе, приведя эквивалентные определениям условия:

- $A \in \mathsf{NP}$?? $\Longleftrightarrow$ существует ли такой набор $\vec{y}$, что $P(\vec{y}) = 1$, где P – функция-описание (условие) задачи. По сути, $\vec{y}$ – это сертификат. $P(y)$ должна работать за полиномиальное от размера описания (входа) время.
- $B \in \mathsf{coNP}$?? $\Longleftrightarrow$ для любого ли набора $\vec{y}$ выполнено, что $Q(\vec{y}) = 1$, где Q – полиномиально проверяемое свойство. Мы перебираем все возможные $\vec{y}$. Если $B = \overline{A}$, то $Q = \neg P$.

SAT и компания

Теорема Кука-Левина

Ранее мы упоминали **Теорему Кука-Левина**. У неё существует альтернативная аналогичная формулировка, которую мы будем доказывать:

Теорема Кука-Левина.

$$SAT \in \mathsf{NPC}$$

Доказательство. Сначала докажем следующую лемму:

Лемма. *Пусть язык (задача)* $\mathscr{A} \in \mathsf{NP}$. *Тогда* $\mathscr{A} \leqslant_p SAT$.

Доказательство. Пусть M – НМТ со временем работы не больше $n^k - 3$ (n – длина входа, k – некое натуральное число), такая, что $\mathscr{L}(M) = \mathscr{A}$.

Далее провернём следующий приём: пусть имеется какая-то конфигурация машины M: $acdb\overset{\nabla q_6}{e}ac\_\_\ldots$ – M находится в состоянии q_6, головка смотрит на e. Преобразуем её к следующему виду:

$$acdb\overset{\nabla q_6}{e}ac\_\_\ldots \Longrightarrow \#acdbq_6eac\_\_\ldots\#$$

– записали всё в одну строку (состояние перед символом, на который указывает головка), и ограничили с обеих сторон символом-разделителем # (который не был в рабочем алфавите МТ изначально).

Построим функцию f, необходимую для сводимости.

Рассмотрим одну ветвь НМТ и будем записывать получаемые конфигурации в специальную таблицу – всего будет не более n^k (время работы не более n^k-3, и "влезет"ветвь любой длины), а столбцов, соответственно, не более $n^k - 3 + 3 = n^k$ (ко входному слову добавляются символ состояния и два символа разделителя #).

Пусть на вход M было подано слово $w = w_1w_2w_3\ldots w_n$. Тогда таблица будет иметь следующий вид:

$$\begin{array}{c} \\ n^k \end{array} \overset{\overset{n^k}{\downarrow}}{\begin{pmatrix} \# & q_0 & w_1 & w_2 & \ldots & w_n & \_ & \_ & \ldots & \ldots & \ldots & \_ & \# \\ \# & p_1 & q_1 & w_2 & \ldots & w_n & \_ & \_ & \ldots & \ldots & \ldots & \_ & \# \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ \# & p_1 & p_2 & \ldots & \ldots & \ldots & p_{m-1} & q_{accept} & p_m & \_ & \ldots & \_ & \# \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \end{pmatrix}}$$

– после окончания слова стоят пробельные символы $\_$.

Функция f будет сначала строить описание всех возможных ветвей МТ (по состояниям) в виде таблицы, со строками, заполненными по принципу выше.

Факт: Слово w находится в языке $\mathscr{A}$ тогда и только тогда, когда таблицу можно заполнить так, чтобы в какой-то из её ячеек появилось q_{accept}

Далее наша функция f будет записывать факт выше в виде булевой формулы. То есть, выдаёт булеву формулу, описывающую все возможные ветви вычисления МТ M на данном входном слове w. Это и позволит нам свести задачу к SAT.

Пусть Q – множество состояний M, Γ – рабочий алфавит M. Множество $C = Q \cup \Gamma \cup \{\#\}$ – символы, которыми мы заполняем таблицу.

На входе МТ у нас всё то же слово w. Мы хотим получить такую булеву формулу φ, что $w \in \mathscr{A} \Longleftrightarrow \varphi \in SAT$. Наша φ будет конъюнкцией 4-х формул:

$$\varphi = \varphi_{cell} \wedge \varphi_{start} \wedge \varphi_{accept} \wedge \varphi_{move}$$

где:

- φ_{cell} – условие, что в каждой ячейке таблицы ровно 1 элемент, причем этот элемент – символ из рабочего алфавита
- φ_{start} – условие, что вход записан правильно
- φ_{accept} – условие, что в какой-то ячейке таблицы есть q_{accept}
- φ_{move} – условие, согласовывающее работу МТ

В нашей булевой формуле будут использоваться следующие переменные: $x_{ijs} = 1$, $i,\ j \in [1,\ n^k]$ означает, что в ячейке $(i,\ j)$ находится символ $s \in C$. Всего будет $O(n^{2k})$ переменных такого вида (по числу занятых ячеек таблицы).

Теперь запишем все 4 формулы при помощи данных переменных:

$$\varphi_{cell} = \bigwedge_{i,\ j \in [1,\ n^k]} \left[\bigvee_{s \in C} x_{ijs} \wedge \bigwedge_{s \neq t \in C} (\neg x_{ijs} \vee \neg x_{ijt}) \right]$$

– берём конъюнкцию по всем ячейкам на условие, что хотя бы 1 символ из C да и есть в ячейке (большая дизъюнкция), но при этом в этой ячейке нет более никакого другого символа (а точнее, есть только этот), кроме данного (конъюнкция с большой конъюнкцией).

$$\varphi_{start} = x_{11\#} \wedge x_{12q_0} \wedge x_{13w_1} \wedge \ldots \wedge x_{1(n+2)w_n} \wedge x_{1(n+3)\_} \wedge \ldots \wedge x_{1(n^k-1)\_} \wedge x_{1n^k\#}$$

– напрямую проверяем, что на вход подали именно то самое входное слово $w = w_1 w_2 w_3 \ldots w_n$ и ни что иное.

$$\varphi_{accept} = \bigvee_{i,\ j \in [1,\ n^k]} x_{ijq_{accept}}$$

– проверяем по всем ячейкам таблицы, что хотя бы в одной из них есть символ q_{accept}.

Перед тем, как записывать в виде булевой формулы φ_{move}, введём понятие **допустимого окна**.

Определение 18. ***Допустимым окном*** – $\begin{array}{|c|c|c|}\hline a & b & c \\ \hline d & e & f \\ \hline \end{array}$ *– будем называть прямоугольный срез таблицы ветви НМТ, такой, что каждая строка среза, начиная со второй, может быть получена за один такт работы данной Машины Тьюринга из предыдущей строки (в соответствии с правилами нашей МТ).*

Примеры допустимых и недопустимых окон :

$\begin{array}{|c|c|c|}\hline a & b & q_1 \\ \hline a & q_2 & b \\ \hline q_3 & a & c \\ \hline \end{array}$ –допустимо, $\begin{array}{|c|c|}\hline \# & a \\ \hline \# & a \\ \hline \end{array}$ – допустимо, $\begin{array}{|c|c|c|c|}\hline a & b & c & q_3 \\ \hline a & q_3 & c & d \\ \hline \end{array}$ –не допустимо,

$\begin{array}{|c|c|c|}\hline q_1 & a & a \\ \hline a & q_2 & a \\ \hline \end{array}$ при отсутствии перехода $(q_1,\ a) \to$ – не допустимо.

Последний пример как раз иллюстрирует случай, когда в целом такое окно могло существовать, но по правилам перехода нашей МТ такой ситуации возникнуть не могло, и окно допустимым не является.

С этого момента будем использовать допустимые окна размерами только 2×3. Теперь будем описывать условия φ_{move}, используя допустимые окна. Пусть верхняя строка нашей таблицы – это начальная конфигурация, и пусть каждое окно в нашей таблице допустимо – иными словами, каждая строка таблицы, начиная со второй – это конфигурация, легально следующая за конфигурацией предыдущей строки.

Рассмотрим два случая, какие ячейки вообще могли получиться, и почему МТ будет работать корректно:

1. Ячейка не граничит с q_i – ячейкой с символом состояния – и не содержит символы состояния. Тогда после одного такта Машины Тьюринга в ячейке всё останется без изменений – соответственно в следующей строке таблицы содержимое данной ячейки ленты останется тем же, и МТ работает корректно.

2. Ячейка граничит с q_i. Тогда рассмотрим окно 2×3: $\begin{array}{|c|c|c|}\hline a & q_i & b \\ \hline c & d & e \\ \hline\end{array}$ – на месте a, b, c, d, e – какие-то символы из C. Ячейка с состоянием в верхней строке посередине. По предположению, такое окно является допустимым. Но тогда и переход с первой строки на вторую будет легален, и Машина работает корректно.

Теперь мы можем записать φ_{move}, используя терминологию допустимых окон:

$$\varphi_{move} = \bigwedge_{\substack{1\leqslant i<n^k \\ 1<j<n^k}} \underset{j}{\square\square\square} \; i - \textbf{допустимые}$$

$$\underset{j}{\square\square\square} \; i - \textbf{допустимые} = \bigvee_{\substack{\begin{array}{|c|c|c|}\hline a & b & c \\ \hline d & e & f \\ \hline\end{array} \\ -\text{допустимое окно}}} \hat{\wedge}\, x$$

$$\hat{\wedge}\, x = x_{i(j-1)a} \wedge x_{ijb} \wedge x_{i(j+1)c} \wedge x_{(i+1)(j-1)d} \wedge x_{(i+1)je} \wedge x_{(i+1)(j+1)f}$$

То есть, мы берём конъюнкцию по дизъюнкциям всех допустимых окон размера 2×3, у которых верхняя строка имеет номер i, а средний столбец – номер j, таких, что эти окна являются допустимыми только при данных значениях этих 6 переменных в окне.

Итак, мы закончили описание второго шага нашей функции f. Условие $w \in \mathscr{A} \iff f(w) = \varphi \in SAT$ выполнено по построению. Осталось показать, что f является вычислимой за полиномиальное время:

- φ_{cell} – так как $|C| = const$, то значит, пройдясь по всем ячейкам за $O(n^{2k})$, мы совершим на каждом шаге лишь константное число операций. Значит, время вычисления φ_{cell} равно $O(n^{2k})$.

- φ_{start} – в первой строке у нас входное слово размера n, и ещё множество пробелов вплоть до конца строки. Тогда время проверки всей строки займёт $O(n^k)$, за которое и вычисляется φ_{start}.

- φ_{accept} – чтобы отыскать q_{accept}, нужно пробежаться по всем ячейкам таблицы, что потребует от φ_{accept} времени $O(n^{2k})$.

- φ_{move} – мы идём по всем допустимым окнам, которые зависят от выбранных (i, j), каждый не более n^k – это даёт минимум $O(n^{2k})$ действий. Далее, в каждом из окон мы ведём перебор содержимого – для каждой из 6 ячеек мы проверяем $|C|$ значений, и всё это вместе. В итоге будет $|C|^6$ вариантов – но это константная величина. Следовательно, ассимптотически это число ни на что не влияет. В итоге получаем время вычисления $O(n^{2k})$.

Как можно заметить, каждая из частей вычисляется за полиномиальное от n время. Ну а так как

$$\varphi = \varphi_{cell} \wedge \varphi_{start} \wedge \varphi_{accept} \wedge \varphi_{move}$$

то для вычисления всего φ мы просто последовательно будем вычислять каждую из 4 частей. Что, опять же, в итоге даст время вычисления $4 \cdot O(n^{2k}) = O(n^{2k})$ – полиномиальное от n – размера входа.

Таким образом, мы привели функцию f, вычислимую за полиномиальное время, такую, что $w \in \mathscr{A} \iff f(w) = \varphi \in SAT$. А это по определению означает, что $\mathscr{A} \leqslant_p SAT$.

Лемма таки доказана! □

Теперь проверим оба условия из определения NP-полной задачи:

1. $SAT \in \mathsf{NP}$, так как сертификатом в алгоритме верификации будет любое выполняющее означивание (вычислимое за полиномиальное от входа время).
2. Согласно лемме выше, $\forall\ \mathscr{A} \in \mathsf{NP} : \mathscr{A} \leqslant_p SAT$, что соответствует определению NP-трудного языка.

Оба условия выполняются, значит, $SAT \in \mathsf{NPC}$. □

Сведения SAT

Задачу SAT можно свести к ещё более простым.

Для начала, представим $\varphi \in SAT$ в виде КНФ, то есть, чтобы вместо любых выполнимых булевых формул у нас были только КНФ. Рассмотрим для этого части φ из предыдущей теоремы. φ_{cell}, φ_{start}, φ_{accept} уже являются КНФ сами по себе. В φ_{move} нужно просто раскрыть скобки в конъюнкции от дизъюнкций допустимых окон, и мы тоже придём к КНФ! Значит, задачу $\phi \in SAT$ можно представить в виде КНФ, не повлияв на время вычисления формулы (всего $O(n^{2k})$ скобок, на каждую потратим константное время).

Утверждение 6. $SAT \leqslant_p 3SAT$

Доказательство. Докажем по определению, что существует такая вычислимая за полиномиальное время функция f, что $\varphi \in SAT \iff f(\varphi) \in 3SAT$.

Пусть $\varphi \in SAT$. Для начала представим её в виде КНФ, что не повлияет на ассимптотику. Достаточно просто предъявить алгоритм построения формулы из $3SAT$ по КНФ из SAT, сделав так, чтобы в скобках с дизъюнкциями было ровно 3 литеры.

Приведём общий алгоритм для различного числа литер в дизъюнкции из SAT:

1. $(l_1) \Longrightarrow (l_1 \vee l_1 \vee l_1)$ – повторяем 3 раза
2. $(l_1 \vee l_2) \Longrightarrow (l_1 \vee l_1 \vee l_2)$ – повторяем какую-то из двух литер ещё раз
3. $(l_1 \vee l_2 \vee l_3) \Longrightarrow (l_1 \vee l_2 \vee l_3)$
4. $(l_1 \vee l_2 \vee l_3 \vee l_4)$ – введём дополнительную переменную ξ –
   $\Longrightarrow (l_1 \vee l_2 \vee \xi) \wedge (l_3 \vee l_4 \vee \neg\xi)$
5. $(l_1 \vee l_2 \vee l_3 \vee l_4 \vee l_5)$ – введём 2 дополнительных переменных ξ_1 и ξ_2 –
   $\Longrightarrow (l_1 \vee l_2 \vee \xi_1) \wedge (l_3 \vee \neg\xi_1 \vee \xi_2) \wedge (l_4 \vee l_5 \vee \neg\xi_2)$
6. $(l_1 \vee l_2 \vee l_3 \vee l_4 \vee l_5 \vee l_6)$ – аналогично случаю для 4 и 5 литер.
7. И так далее...

Корректность переходов для 1, 2 и 3 литер очевидна. Корректность перехода для 4 литер мы доказывали в предыдущем разделе. Корректность переходов для пяти и более литер доказывается аналогично случаю для 4 литер.

Как мы видим, построение $f(\varphi) \in 3SAT$ по $\varphi \in SAT$ вида КНФ занимает полиномиальное от размера φ время. В совокупности с тем, что перевод φ в КНФ также занимает полиномиальное от размера φ время, можно заключить, что функция f является вычислимой за полиномиальное время.

Таким образом, существует такая вычислимая за полиномиальное время функция f, что $\varphi \in SAT \iff f(\varphi) \in 3SAT$. А это по определению означает, что $SAT \leqslant_p 3SAT$. □

NPC задачи и SAT

Задача о сумме подмножества

Рассмотрим теперь другую задачу, совершенно непохожую на SAT на первый взгляд – **Задача сумме подмножества**, или $Subset - Sum$.

Subset-Sum

Ввод: числа $(x_1, \ldots, x_q)$, число $T \in \mathbb{N}$

Вывод: Ответ на вопрос: "существует ли подмножество индексов $S \subseteq \{1, \ldots, q\}$, такое, что $\sum\limits_{i \in S} x_i = T$??"

Теорема 3. $Subset - Sum \in \mathsf{NPC}$

Доказательство. Для начала докажем следующую лемму:

Лемма. $3SAT \leqslant_p Subset - Sum$

Доказательство. Для начала, приведём такую функцию f, которая за полиномиальное время будет сводить задачу $3SAT$ к задаче о сумме подмножества, то есть:

$$(l_1^1 \vee l_2^1 \vee l_3^1) \wedge \ldots \wedge (l_1^n \vee l_2^n \vee l_3^n) \xrightarrow{f} (x_1, \ldots, x_q), T$$

Пусть y_i – переменная в $3SAT$. Сопоставим каждому y_i две новых переменных: $a_i := y_i$ и $b_i := \neg y_i$. Иначе говоря, $\begin{cases} a_i \text{ истинно} \iff y_i \text{ истинно} \\ b_i \text{ истинно} \iff y_i \text{ ложно} \end{cases}$

Мы хотим, чтобы в подмножестве $(x_1, \ldots, x_q)$ была от каждого y_i выбрана ровно 1 переменная. Какими тогда должны быть a_i, b_i, T?

Сделаем следующее. Пусть имеются переменные $y_1, \ldots, y_k$. Построим следующую таблицу, которая представляет собой соответствие между означиванием $3SAT$ и суммой.

В ней будет сначала k (по числу переменных) столбцов – назовём их столбцами значений, потом ещё n (по числу скобок) столбцов – назовём их столбцами скобок. В строках мы будем последовательно записывать переменные a_1, b_1, a_2, b_2, $\ldots$, a_k, b_k, соотвествующие $y_1, \ldots, y_k$. Для каждого a_i и b_i в таблице в i-х столбцах значений будут записаны единицы (1). Для всех $j < i$ в столбцах значений будут записаны 0, для всех $j > i$ в столбцах значений не будет записано ничего. В столбцах скобок мы для каждой переменной запишем символ $*$, такой, что: $* := \begin{cases} 1, \text{ если данная переменная есть в этой скобке} \\ 0, \text{ если данной переменной нет в этой скобке} \end{cases}$

Далее, запишем в строки по 2 переменные на каждую скобку: c_{i1} и c_{i2}. В итоге получится c_{11}, c_{12}, $\ldots$, c_{n1}, c_{n2} – $2n$ дополнительных переменных. В таблице для c_{i1} и c_{i2} мы для обеих в i-х столбцах скобок будут записаны единицы (1). Для всех $j > i$ в столбцах скобок будут записаны 0, для всех $j < i$ в столбцах скобок не будет записано ничего. В столбцах значений у данных переменных пустота.

В последней строке запишем число T. Во всех столбцах значений для T пишем 1, во всех столбцах скобок для T пишем 3.

В общем виде таблица будет выглядеть так:

⇓⇓⇓⇓⇓
⇓⇓⇓⇓⇓
⇓⇓⇓⇓⇓
⇓⇓⇓⇓⇓
⇓⇓⇓⇓⇓
⇓⇓⇓⇓⇓

Переменные $\downarrow$ \ Разряды $\rightarrow$	k	$k-1$	$\ldots$	3	2	1	1	2	$\ldots$	$n-1$	n
a_1			$\ldots$			1	$*$	$*$	$\ldots$	$*$	$*$
b_1			$\ldots$			1	$*$	$*$	$\ldots$	$*$	$*$
a_2			$\ldots$		1	0	$*$	$*$	$\ldots$	$*$	$*$
b_2			$\ldots$		1	0	$*$	$*$	$\ldots$	$*$	$*$
a_3			$\ldots$	1	0	0	$*$	$*$	$\ldots$	$*$	$*$
b_3			$\ldots$	1	0	0	$*$	$*$	$\ldots$	$*$	$*$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
a_{k-1}		1	$\ldots$	0	0	0	$*$	$*$	$\ldots$	$*$	$*$
b_{k-1}		1	$\ldots$	0	0	0	$*$	$*$	$\ldots$	$*$	$*$
a_k	1	0	$\ldots$	0	0	0	$*$	$*$	$\ldots$	$*$	$*$
b_k	1	0	$\ldots$	0	0	0	$*$	$*$	$\ldots$	$*$	$*$
c_{11}			$\ldots$				1	0	$\ldots$	0	0
c_{12}			$\ldots$				1	0	$\ldots$	0	0
c_{21}			$\ldots$					1	$\ldots$	0	0
c_{22}			$\ldots$					1	$\ldots$	0	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$c_{(n-1)1}$			$\ldots$						$\ldots$	1	0
$c_{(n-1)2}$			$\ldots$						$\ldots$	1	0
c_{n1}			$\ldots$						$\ldots$		1
c_{n2}			$\ldots$						$\ldots$		1
T	1	1	$\ldots$	1	1	1	3	3	$\ldots$	3	3

В частности, для булевой формулы

$$(y_1 \vee \neg y_2 \vee \neg y_3) \wedge (\neg y_1 \vee \neg y_2 \vee y_3)$$

таблица получится следующей:

Переменные $\downarrow$ \ Разряды $\rightarrow$	3	2	1	1	2
a_1			1	1	0
b_1			1	0	1
a_2		1	0	0	0
b_2		1	0	1	1
a_3	1	0	0	0	1
b_3	1	0	0	1	0
c_{11}				1	0
c_{12}				1	0
c_{21}					1
c_{22}					1
T	1	1	1	3	3

Теперь объясним, а во имя чего, собственно, строить такую таблицу. Суть в том, что числа по строкам, что получились из a_i, b_i и c_{j1} с c_{j2} – это и есть наши числа $(x_1, \ \ldots, \ x_q)$, а получившееся число T внизу таблицы – это и есть то самое T, для которого необходимо найти суммирующее подмножество. Нам необходимо выбрать для каждой переменной y_i одно из чисел a_i или b_j и только одно. Их мы будем складывать. Так как для y_i у a_i и b_i единица стоит только в i-м столбце значений, то следовательно, при сложении мы в числе T получим в начале ровно k единиц. В $3SAT$ в каждой скобке по 3 литеры. Это значит, что в каждом из последних n разрядов у числа T будет стоять либо цифра 1 (как минимум, ведь булева

формула должна быть выполнимой, раз она из языка $3SAT$), либо 2, либо 3 (максимум, так как в каждой скобке по 3 литеры) – в зависимости от того, в каких скобках есть a_i или b_j. Мы хотим, чтобы в последних n разрядах в каждом суммарно вышло по 3. Для этого и заводятся c_{ij}, чтобы для каждой скобки докинуть от 0 до 2 дополнительных переменных, увеличив тем самым сумму в разряде до 3.

В итоге мы получаем числа $a_i = 10^{i-1} \cdot 10^n + d_{ai}$, $b_i = 10^{i-1} \cdot 10^n + d_{bi}$, где числа d_{ai} и d_{bi} представляют из себя как раз те части из $*$. Ещё мы имеем числа c_{j1} и c_{j2}, каждое из которых равно 10^{n-j}. Также есть число T, которое в первых k разрядах имеет 1, в последних n разрядах имеет 3. Тогда:

$$\{x_1,\, x_2,\, \ldots,\, x_q\} = \{a_1,\, \ldots a_k,\, b_1,\, \ldots,\, b_k,\, c_{11},\, c_{12},\, \ldots,\, c_{n1},\, c_{n2}\}$$
$$T = \underbrace{11\ldots11}_{k}\underbrace{33\ldots33}_{n}$$

Тем самым мы свели задачу к $Subset - Sum$. Размер таблицы получился $(2n+2k+1) \times (n+k)$ – полиномиальный от размера булевой формулы на входе функции f.

Покажем теперь, что сведение корректно, то есть, $\varphi \in 3SAT \iff f(\varphi) \in Subset - Sum$:

- $\varphi \in 3SAT \Longrightarrow f(\varphi) \in Subset - Sum$

  Пусть $\varphi \in 3SAT$ выполнима. Тогда существует выполняющее означивание. Рассмотрим его. Для каждой пары литер y и $\neg y$ только одна из них истинна. Тогда, если $y_i = 1$, то выбираем к сумме a_i, если $y_i = 0$, то выбираем к сумме b_i. После этого в старших разрядах суммы (числа T мы как раз получим k единиц).

  Так как означивание выполняющее, то в каждой скобке 1, 2 или 3 литеры истинны. Тогда в последних разрядах T получим 1, 2 или 3. Если получили 3, то не добираем c_{j1} или c_{j2}. Если 2 – докидываем к набору, образующему сумму, одно из c_{j1} и c_{j2}. Если 1, то добавляем оба. Таким образом в сумме мы получили 3 в разряде.

  Это означает, что существует такое подмножество (которое мы набрали) чисел a_i, b_i и c_{j1} с c_{j2}, что в сумме дают $T = \underbrace{11\ldots11}_{k}\underbrace{33\ldots33}_{n}$, и задача лежит в $Subset - Sum$.

- $f(\varphi) \in Subset - Sum \Longrightarrow \varphi \in 3SAT$

  Пусть в таблице, что мы получили, у нас существует подмножество чисел a_i, b_i и c_{j1} с c_{j2}, что в сумме дают число T из таблицы же. Заметим, что в каждом столбце скобок находится ровно 5 единиц – так как в скобках у нас было по 3 литеры, к каждой скобке ещё по единичке от c_{j1} и c_{j2}. Это означает, что в итоговой сумме нет переносов разряда, и сумма корректна – если написано 3, то суммируем 3 числа, если написано 1, то только одно число. Рассмотрим теперь a_i и b_i. Если в наборе есть a_i, то значит, $y_i = 1$. Если в наборе есть b_i, то значит, $y_i = 0$. Следовательно, получаем некое означивание переменных $y_1,\, \ldots,\, y_k$.

  Покажем, что оно является выполняющим. Пусть $(l_1^j \vee l_2^j \vee l_3^j)$ – j-я скобка. В j-м столбце скобок в строке числа T стоит 3 – значит, было выбрано 3 числа. Так как из чисел c_{j1} и c_{j2} мы могли выбрать максимум 2, то следовательно, было выбрано хотя бы одно число из множеств a_i и b_i, которе имеет 1 в j-м столбце скобок, и хотя бы одна из литер l_1^j, l_2^j, l_3^j истинна. Но тогда и вся дизъюнкция истинна. Так для каждой скобки. В итоге получаем, что действительно существует выполняющее означивание, по которому строится набоо суммирования числа T.

Таким образом, мы привели функцию f, вычислимую за полиномиальное время, такую, что $\varphi \in 3SAT \iff f(\varphi) \in Subset - Sum$. А это по определению означает,
что $3SAT \leqslant_p Subset - Sum$. □

Теперь проверим оба условия из определения NP-полной задачи:

1. $Subset - Sum \in \mathsf{NP}$, так как сертификатом в алгоритме верификации будет просто набор индексов, таких, что соответствующие числа образуют в сумме T. Суммировать числа и сравнивать мы можем за полиномиальное время.

2. Согласно лемме выше, $3SAT \leqslant_p Subset - Sum$, а так как $3SAT \in \mathsf{NPC}$, то значит, $\forall\ \mathscr{A} \in \mathsf{NP} : \mathscr{A} \leqslant_p 3SAT \leqslant_p Subset - Sum$, что соответствует определению NP-трудного языка.

Оба условия выполняются, значит, $Subset - Sum \in \mathsf{NPC}$. □

Задача о рюкзаке

Вспомним задачу о рюкзаке:
Knapsack

Ввод: числа $(w_1,\ \ldots,\ w_q)$ – веса предметов
числа $(v_1,\ \ldots,\ v_q)$ – стоимости предметов
W – максимальный вес рюкзака
Вывод: Набор предметов как можно большей суммарной стоимости при данных ограничениях на вес

Переформулируем задачу следующим образом:

Ввод: числа $(w_1,\ \ldots,\ w_q)$ – веса предметов
числа $(v_1,\ \ldots,\ v_q)$ – стоимости предметов
W – максимальный вес рюкзака
V – стоимость
Вывод: Ответ на вопрос: "Существует ли подмножество индексов $S \subseteq \{1,\ \ldots,\ q\}$, такое, что $\sum\limits_{i \in S} w_i \leqslant W$ и $\sum\limits_{i \in S} v_i \geqslant V$??"

Будем теперь рассматривать задачу о рюкзаке в данной формулировке.

Теорема 4. *$Knapsack \in \mathsf{NPC}$.*

Доказательство. Для начала докажем следующую лемму:

Лемма. *$Subset - Sum \leqslant_p Knapsack$*

Доказательство. Приведём функцию f, преобразующую задачу о сумме подмножества к задаче о рюкзаке напрямую за полиномиальное время.

Пусть имеется множество чисел $\{x_1, x_2,\ \ldots, x_q\}$ и число T, которое надо суммировать. Тогда сделаем q предметов с весами $x_1, x_2,\ \ldots, x_q$ соотвественно, стоимостями тоже $x_1, x_2,\ \ldots, x_q$ соответственно. Пускай максимальный вес рюкзака будет равен числу T, и ему же пусть будет равна стоимость V. Все эти действия делаются за полином.

Тогда получаем задачу о рюкзаке с условием: "Существует ли подмножество индексов $S \subseteq \{1,\ \ldots,\ q\}$, такое, что $\sum\limits_{i \in S} x_i \leqslant T$ и $\sum\limits_{i \in S} x_i \geqslant T$??". Очевидно, что оно эквивалентно условию задачи о сумме подмножества.

Таким образом, мы привели функцию f, вычислимую за полиномиальное время, такую, что переводит задачу о сумме подмножества в эквивалентную задачу о рюкзаке. А это по определению означает, что $Subset - Sum \leqslant_p Knapsack$. □

Теперь проверим оба условия из определения NP-полной задачи:

1. $Knapsack \in \mathsf{NP}$, так как сертификатом в алгоритме верификации будет просто набор индексов, таких, что соответствующие предметы по весам и стоимостям удовлетворяют ограничениям на W и V. Суммировать числа и сравнивать мы можем за полиномиальное время.

2. Согласно лемме выше, $Subset - Sum \leqslant_p Knapsack$, а так как $Subset - Sum \in \mathsf{NPC}$, то значит, $\forall\ \mathscr{A} \in \mathsf{NP} : \mathscr{A} \leqslant_p Subset - Sum \leqslant_p Knapsack$, что соответствует определению NP-трудного языка.

Оба условия выполняются, значит, $Knapsack \in \mathsf{NPC}$. □

Задача о поиске максимального разреза

Вспомним задачу о поиске максимального разреза:
Max-Cut

Ввод: $G = (V, E)$, $k \in \mathbb{N}$ – невзвешенный неориентированный граф

Вывод: Ответ на вопрос: "Существует ли (A, B) – разбиение вершиин графа на два множества, такое, что $A \cup B = V$, $A \cap B = \varnothing$, $A \neq \varnothing$, $B \neq \varnothing$, причём количество рёбер, соединяющих вершины из разных множеств – $|\{(u, v) \mid u \in A, v \in B\}| \geqslant k$??"

Теорема 5. $Max - Cut \in \mathsf{NPC}$

Доказательство. Для начала докажем следующую лемму:

Лемма. $NAE - 3SAT \leqslant_p Max - Cut$

Доказательство. Приведём функцию f, вычислимую за полиномиальное время, которая сводит задачу $NAE - 3SAT$ к задаче о поиске максимального разреза, строя необходимый граф по булевой формуле.

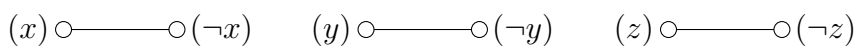
Пускай у нас имеется булева формула на l переменных и n скобок. Будем считать, сколько у нас вышло рёбер в разрезе. Опишем общий алгоритм, параллельно показывая на примере. Пускай имеется формула $\varphi \in NAE - 3SAT$:

$$(x \circ \neg y \circ z) \wedge (\neg x \circ \neg y \circ \neg z)$$

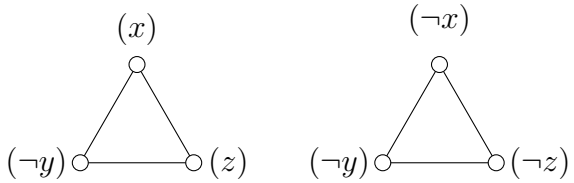
Идея построения графа заключается в том, чтобы в получившемся максимальном разрезе $(\mathcal{T}, \mathcal{F})$ с одной стороны были все истинные литеры, а с другой – все ложные.

Для начала, каждой переменной p задаём 2 вершины – (p) и $(\neg p)$ – и соединяем их ребром:

Обозначим такую пару вершин, соединённых ребром, **гаджет-переменной**. Всего l переменных, значит, проведём l рёбер. Тогда для нашей булевой формулы выйдет такая конструкция:

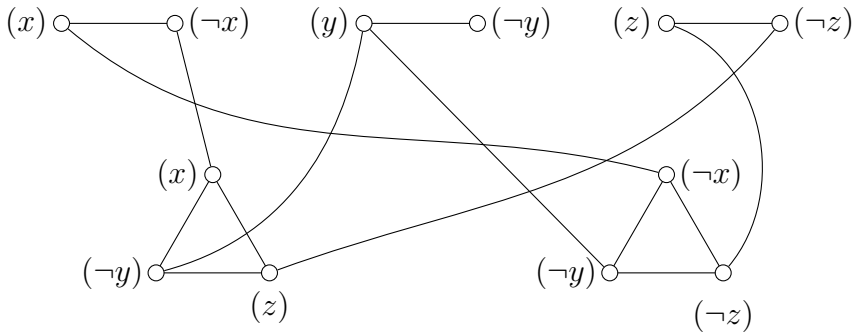


Затем, для каждой скобки мы делаем по «треугольнику», создавая по вершине на каждую литеру скобки и соединяя их друг с другом рёбрами.



Обозначим такой треугольник из трёх вершин, попарно соединённых рёбрами, **гаджет-скобкой**. Всего у нас n скобок, и каждая скобка будет либо полностью входить в какую-то часть разреза (если все литеры одинаковы), либо одна вершина в одной части, две оставшиеся в другой. В последенем случае разрез пересекают 2 ребра, и все гаджет-скобки добавят $2n$ рёбер к разрезу.

Далее, для каждой вершины гаджет-скобки делаем следующее: соединяем её ребром с вершинной гаджет-переменной, такой, что отвечает противоположному значению литеры данной вершины. То есть, (p) соединяем с $(\neg p)$, и соответственно $(\neg p)$ соединяем с (p):



От каждой гаджет-скобки мы проводим по 3 ребра, соединяющие вершины гаджет-скобки с «противоположными» вершинами гаджет-переменных. Следовательно, все рёбра будут пересекать разрез, что даёт ещё $3n$ рёбер.

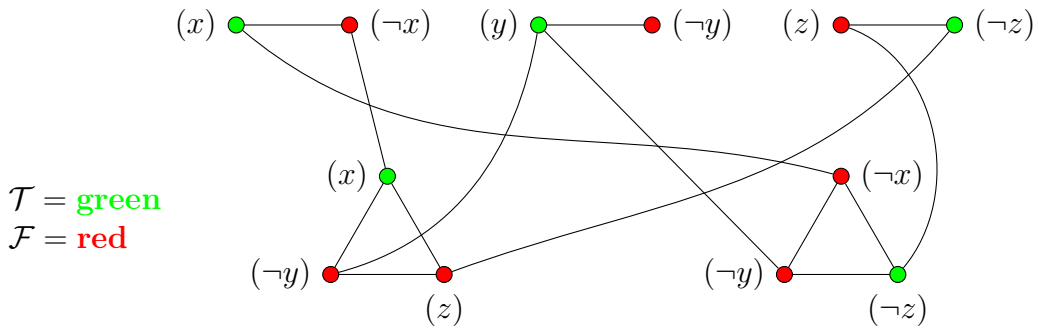
Получили граф. Подсчитаем количество рёбер разреза графа, тем самым оценив время работы f.

Наша функция строит разрез $(\mathcal{T}, \mathcal{F})$ так, чтобы все истинные литеры были в одной части разреза – в $\mathcal{T}$, а все ложные – в другой, $\mathcal{F}$. Каждая гаджет-переменная добавляет к разрезу по ребру, каждая гаджет-скобка – по 2 ребра, и соединение гаджет-скобок с гаджет-переменными даст нам ещё по 3 ребра на гаджет скобку. Тогда всего получается $l + 2n + 3n = l + 5n$ рёбер, значит, разрез имеет полиномиальный от входа размер, и наша функция f является вычислимой за полиномиальное время.

Покажем теперь, что сведение корректно, то есть, $\varphi \in NAE{-}3SAT \iff f(\varphi) \in Max{-}Cut$:

- $\varphi \in NAE - 3SAT \implies f(\varphi) \in Max - Cut$

  Пусть φ – выполнимая $NAE - 3SAT$. Тогда возьмём выполняющее означивание и рассмотрим разрез $(\mathcal{T}, \mathcal{F})$, где в $\mathcal{T}$ входят все вершины, отвечающие истинным литерам, а в $\mathcal{F}$ все, отвечающие ложным. Например, для нашего примера выполняющим будет означивание $x = 1$, $y = 1$, $z = 0$, и граф с разрезом примет вид:



  Покажем, что размер полученного разреза равен в точности $l + 5n$.

  Каждая гаджет-переменная к размеру разреза добавит по одному ребру, так как только одна литера из p и $\neg p$ может быть истинной, вторая уже ложна. Значит, l рёбер от гаджет переменных.

  Далее, так как $\varphi \in NAE{-}3SAT$ выполнима, то в каждой скобке в точности одна литера отлична от двух других. Значит, в гаджет-скобке какая-то вершина будет в одной части разреза, а две оставшиеся в другой. Тогда гаджет-скобка даст 2 ребра к размеру разреза.

Значит, так как всего у нас n скобок и каждая истинна, то соотвественно n гаджет-скобок, которые добавят $2n$ рёбер к размеру разреза.

После этого мы соединили вершины в гаджет-скобках с вершинами в гаджет-переменных. Так как мы соединяли противоположные по значению литеры, то каждое ребро будет пересекать разрез. Всего у нас n гаджет-скобок, и от каждой мы провели 3 ребра, принадлежащие разрезу. Это даёт $3n$ рёбер.

Всего получается $l + 2n + 3n = l + 5n$ рёбер. Следовательно, мы получили разрез максимального размера, и $(\mathcal{T}, \mathcal{F}) = f(\varphi) \in Max - Cut$.

- $f(\varphi) \in Max - Cut \Longrightarrow \varphi \in NAE - 3SAT$

  Пусть имеется максимальный разрез $(\mathcal{T}, \mathcal{F}) = f(\varphi) \in Max-Cut$ размера $l+5n$. Докажем, что в таком случае φ выполнима.

  Покажем для начала корректность разреза, то есть, что он берёт по ребру от гаджет-переменных, по 2 ребра от гаджет-скобок и по 3 ребра от соединений гаджетов.

  Через разрез не могут одновременно проходить все три ребра гаджет-скобки в силу того факта, что граф двумерен. Значит, либо все 3 вершины по одну сторону разреза, либо две вершины в одной части разреза, и оставшаяся вершина в другой. Значит, рёбра гаджет-скобки либо вообще не пересекают разрез, либо ровно 2 ребра его пересекают. Следовательно, так как всего n гаджет-скобок, может быть не более $2n$ рёбер от них, принадлежащих разрезу.

  В таком случае, у нас остаётся не менее $l + 3n$ рёбер, которые **обязаны** пересечь разрез, чтобы получить необходимый необходимый размер – $l + 5n$. Следовательно, оставшиеся $l + 3n$ рёбер – те, что от гаджет-переменных, и те, что от соединений гаджетов – точно пересекают разрез. Но тогда выходит, что разрез берёт по ребру от каждой из l гаджет-переменных и по 3 ребра от каждого из n соединений гаджетов. Тогда оставшимся $2n$ рёбрам разреза ничего не остаётся, кроме как принадлежать гаджет-скобкам. Значит, разрез корректен.

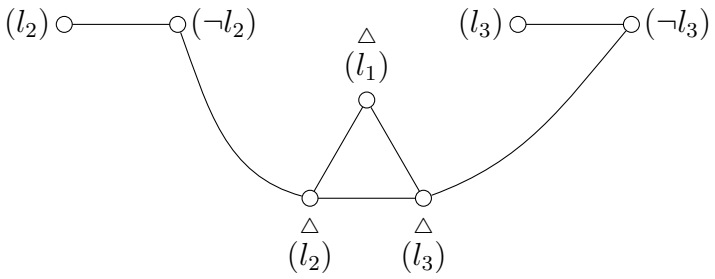
  Теперь построим выполняющее означивание по разрезу, чтобы доказать выполнимость φ.

  Имеем разрез $(\mathcal{T}, \mathcal{F})$. Для каждого ребра из разреза мы знаем, что одна из вершин ребра принадлежит $\mathcal{T}$, а другая – $\mathcal{F}$.

  Действуем: $\begin{cases} \text{все чистые переменные литер вершин из } \mathcal{T} \text{ делаем истинными } (1) \\ \text{все чистые переменные литер вершин из } \mathcal{F} \text{ делаем ложными } (0) \end{cases}$

  Покажем, что это даст нам необходимое означивание, то есть, что в каждой скобке есть хотя бы одна истинная литера и хотя бы одна ложная.

  Рассмотрим какую-то гаджет-скобку:



  Без ограничения общности, положим $(l_2) \in \mathcal{T}$, $(l_3) \in \mathcal{F}$. Они соединены рёбрами с $(\neg l_2)$ и $(\neg l_3)$ соответственно. Рёбра гаджет-переменных принадлежат разрезу, значит, $(\neg l_2) \in \mathcal{F}$ и

$(\neg l_3) \in \mathcal{T}$. Посмотрим на вершину гаджет-скобки $(\overset{\triangle}{l_2})$. Она соединена ребром с вершиной гаджет-переменной – $(\neg l_2)$. Как мы установили ранее, это ребро из разреза, следовательно, $(\overset{\triangle}{l_2})$ и $(\neg l_2)$ в разных частях разреза. Так как $(\neg l_2) \in \mathcal{F}$, то значит $(\overset{\triangle}{l_2}) \in \mathcal{T}$. Посмотрим на вершину гаджет-скобки $(\overset{\triangle}{l_3})$. Она соединена ребром с вершиной гаджет-переменной – $(\neg l_3)$. Как мы установили ранее, это ребро из разреза, следовательно, $(\overset{\triangle}{l_3})$ и $(\neg l_3)$ в разных частях разреза. Так как $(\neg l_3) \in \mathcal{T}$, то значит $(\overset{\triangle}{l_3}) \in \mathcal{F}$.

Значит, для этой гаджет скобки $(\overset{\triangle}{l_2}) \in \mathcal{T}$ и $(\overset{\triangle}{l_3}) \in \mathcal{F}$ – есть две вершины, принадлежащие разным частям разреза. Тогда соответственно в скобке формулы φ, соответсвующей данной гаджет-скобке – $(l_1 \circ l_2 \circ l_3)$ – будет хотя бы одна истинная литера – l_2 – и хотя бы одна ложная литера – l_3. В таком случае, $(l_1 \circ l_2 \circ l_3)$ истинна.

Получается, что при вышеуказанном действии (все из $\mathcal{T}$ равны 1, все из $\mathcal{F}$ равны 0) все скобки φ будут истинны. Следовательно, это даст нам выполняющее означивание, откуда $\varphi \in NAE - 3SAT$.

Таким образом, мы привели функцию f, вычислимую за полиномиальное время, такую, что $\varphi \in NAE - 3SAT \iff f(\varphi) \in Max - Cut$. А это по определению означает,
что $NAE - 3SAT \leqslant_p Max - Cut$. □

Теперь проверим оба условия из определения NP-полной задачи:

1. $Max - Cut \in \mathsf{NP}$, так как сертификатом в алгоритме верификации будет сам разрез, то есть, два набора вершин. Проверка рёбер между наборами занимает полиномиальное от размера входа время.

2. Согласно лемме выше, $NAE - 3SAT \leqslant_p Max - Cut$, а так как $NAE - 3SAT \in \mathsf{NPC}$, то значит, $\forall\ \mathscr{A} \in \mathsf{NP} : \mathscr{A} \leqslant_p NAE - 3SAT \leqslant_p Max - Cut$, что соответствует определению NP-трудного языка.

Оба условия выполняются, значит, $Max - Cut \in \mathsf{NPC}$. □

TIME и языковая иерархия

Классы TIME

Определение 19. $\mathsf{TIME}\big(t(n)\big)$ – *множество языков, разрешимых за время* $O\big(t(n)\big)$ *на Детерминированной МТ.*

Определение 20. $\mathsf{NTIME}\big(t(n)\big)$ – *множество языков, разрешимых за время* $O\big(t(n)\big)$ *на Недетерминированной МТ.*

Через TIME и NTIME можно переопределить понятния P и NP:

Определение. $\mathsf{P} = \bigcup\limits_{k\in\mathbb{R}} \mathsf{TIME}(n^k)$

Определение. $\mathsf{NP} = \bigcup\limits_{k\in\mathbb{R}} \mathsf{NTIME}(n^k)$

Дадим определение ещё одному классу языков.

Определение 21. $\mathsf{EXPTIME}\big(t(n)\big)$ – *множество языков, разрешимых за экспоненциальное –* $O\big(2^{t(n)}\big)$ *– время на ДМТ.*

Иначе говоря, $\mathsf{EXPTIME} = \bigcup\limits_{k\in\mathbb{R}} \mathsf{TIME}\big(2^{O(n^k)}\big)$.

Выполняется следующее включение: $\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{EXPTIME}$, и вполне может быть, что либо $\mathsf{P} = \mathsf{NP}$, либо $\mathsf{NP} = \mathsf{EXPTIME}$, но это пока что неизвестно.

Вспомним, что NP – класс задач с полиномиальным алгоритмом верификации, за полиномиальное время проверяющим какой-то сертификат. Так вот: EXPTIME язык перебирает все возможные сертификаты, то есть, $|\Sigma|^{n^k}$ сертификатов. Таким образом, все NP-языки можно разрешить за EXPTIME на ДМТ.

Определение 22. *Функция* $t : \mathbb{N} \to \mathbb{N}$*, такая что* $t(n) \geq n \log n$*, называется* ***конструируемой по времени****, если существует ДМТ, перерабатывающая слово из* n *единиц в двоичное представление* $t(n)$ *за* $O\big(t(n)\big)$ *шагов.*

Примеры $t(n)$: $n \log n$, $n\sqrt{n}$, n^2 и подобные.

Такая ДМТ работет по следующему принципу: сначала она перерабатывает слово из n единиц в двоичное представление числа n, а уже затем вычисляет $t(n)$. Приведём эффективный алгоритм перевода слова 1^n в двоичное представление n.

Пусть у нас есть слово $\underbrace{11\ldots11}_{n}$ на ленте ДМТ. Действуем так:

1. Заводим двоичный счётчик в свободной ячейке ленты.
2. Считаем единицы. При прочтении одной единицы увеличиваем счётчик на 1.

Стоит отметить, что если просто держать счётчик в каком-то месте ленты, то при прочтении очередной единицы придётся бежать до него, потом обратно. И так много раз, что довольнотаки долго. Поэтому будем «таскать счётчик за собой»:

- Заводим счётчик прямо перед началом входного слова.
- При прочтении единицы стираем её, увеличиваем счётчик на 1 (с изменением цифр).
- Каждый раз, когда между счётчиком и оставшейся частью входного слова возникает свободная ячейка, сдвигаем счётчик на ячейку вправо, чтобы он опять оказался в начале слова.

Покажем работу алгоритма на примере:

$$\overset{\triangledown}{\circ}||1|1|1|1|1 \Rightarrow \overset{\triangledown}{1}||1|1|1|1 \Rightarrow 1\overset{\triangledown}{0}||1|1|1 \Rightarrow 11\overset{\triangledown}{\circ}||1|1$$
$$\Downarrow \{ \text{ сдвиг } \} \Downarrow$$
$$\circ 1\overset{\triangledown}{1}||1|1 \Rightarrow 10\overset{\triangledown}{0}||1 \Rightarrow 101\overset{\triangledown}{\circ}||$$
$$\Downarrow \{ \text{ сдвиг } \} \Downarrow$$
$$\circ 10\overset{\triangledown}{1}||$$

Можно оценить время работы алгоритма. Всего он совершает n шагов, и каждые 2 шага приходится сдвигать счётчик, на ДМТ тратит $\log n$ времени. Тогда общее время работы будет $O(n \log n)$, и алгоритм работает практически за линейное время.

Далее уже из двоичного представления n мы можем считать $t(n)$.

Теорема об иерархии задач по времени

Теорема об иерархии задач по времени. *Пусть t – функция, конструируемая по времени. Тогда существует язык $\mathscr{L}$, разрешимый за время $O\big(t(n)\big)$, но не за время $o\left(\frac{t(n)}{\log t(n)}\right)$.*

Доказательство. Нужно построить машину Тьюринга D, работающую в течение времени $O\big(t(n)\big)$ и разрешающей язык $\mathscr{L}$, неразрешимый за время $o\left(\frac{t(n)}{\log t(n)}\right)$. Чтобы не переработать, машина должна вычислить значение $t(n)$ и следить за тем, чтобы не выйти за ограничения по времени. Заметим, что D должна быть одноленточной – иначе рискуем выйти за ограничение в силу различных полиномиальных оценок работы.

Пусть наша МТ D получила на вход слово w. Будем действовать по следующему алгоритму:

1. Заведём на ленте МТ переменную $n = |w|$.
2. Вычислим $t(n)$ и сохраним значение $\left\lceil \frac{t(n)}{\log t(n)} \right\rceil$ в двоичном счетчике на ленте. Будем уменьшать счетчик перед каждым следующим шагом МТ. Если на каком-то шаге счётчик достигнет 0, «отвергаем» входное слово, переходя в q_{reject}.
3. Теперь наша МТ будет воспринимать входное слово следующим образом: сначала отбросим все нули в конце слова, пока в конце не окажется единица. После чего убираем и эту единицу. То бишь поделим слово так: $w = \langle M \rangle 10^*$. Смотрим на слово M, оставшееся на ленте. Если M – код некоторой Машины Тьюринга, то продолжаем работу. Если M таковым не является, то переходим в q_{reject}.
4. Имитируем работу Машины M на слове w (на полном слове w).
5. Если в результате работы на слове w машина M перешла в состояние q_{accept}, то наша машина D говорит «нет», переходя в состояние q_{reject}. Если же $M(w) = q_{reject}$, то говорим «да» и переходим в q_{accept}.

Такая машина D существует попросту по построению – можно «запрограммировать» на МТ шаги 1 – 5. Все шаги, кроме третьего, выполняются вполне естественно. Рассмотрим подробнее, как именно машина D имитирует M на входе w.

По факту, нужно хранить 3 вещи:

1. *что* находится на ленте M
2. *как именно* работает МТ M

3. *счётчик времени*

Идея заключается в том, что у нашей D будет 3 дорожки на ленте, которые соотвественно хранят:

1. всю информацию на ленте M
2. текущее состояние и функции переходов M
3. счётчик шагов

Для того, чтобы хранить все эти дорожки на одной ленте, интуитивно хочется попросту поделить ленту на 3 части, обособленные разделителями. Однако, головке МТ придётся тогда постоянно перебегать между тремя частями по многу раз. Сделаем следующее: поделим всю ленту на тройки ячеек: | 1 | 2 | 3 | В первых ячейках троек мы храним информацию на ленте, во вторых – состояния и переходы, в третьих – счётчик. При перемещении головки машины M информация на второй дорожке (2-е ячейки троек) и третьей дорожке (3-и ячейки троек) сдвигается в том же направлении.

Покажем, что это позволяет нам уложиться во временные ограничения, рассмотрев подробнее вторую дорожку:

- Размер информации на второй дорожке зависит только от M (а не от слова, поступающего ей на вход), значит, передвижение занимает константное время.
- Время, затрачиваемое на имитацию работы M, – $O\big(g(n)\big)$, где $g(n)$ – время работы M.

Теперь поговорим подробнее о третьей дорожке. Максимальное значение счетчика равно $\left\lceil \frac{t(n)}{\log t(n)} \right\rceil$. Счётчик занимает всего $\log\left(\left\lceil \frac{t(n)}{\log t(n)} \right\rceil\right) = O\big(\log t(n)\big)$ ячеек, откуда общее время работы будет $O\big(t(n)\big)$.

Теперь покажем, что для разрешения языка $\mathscr{L}$ не будет достаточно времени $\left\lceil \frac{t(n)}{\log t(n)} \right\rceil$.

Пусть Пусть M – машина Тьюринга, разрешающая $\mathscr{L}$ за время $g(n) = o\left(\frac{t(n)}{\log t(n)}\right)$. Тогда, подав D на вход слово $w = \langle M \rangle 10^*$, машине D для имитации работы M потребуется $O\big(g(n)\big) = c{\cdot}g(n)$ времени, где c – константа. Но тогда, по определению, $\exists\, n_0 \in \mathbb{N} \mid \forall\, n \geq n_0 :\ cg(n) < \frac{t(n)}{\log t(n)}$.

За это время D успевает полностью считать вход и сымитировать машину M на входном слове $w = \langle M \rangle 10^{n_0}$ и выдать ответ, противоположный ответу M (по построению нашей МТ, ведь счётчик ещё не обнулился).

Но таким образом, на слове из $\mathscr{L} \in \mathscr{L}(M)$ машина D пришла в q_{reject}, откуда следует, что $\mathscr{L}$ неразрешим за $\left\lceil \frac{t(n)}{\log t(n)} \right\rceil$.

□

Следствие 1. *Для любых двух функций t_1, $t_2 :\ \mathbb{N} \to \mathbb{N}$, таких что $t_1(n) = o\left(\frac{t_2(n)}{\log t_2(n)}\right)$ и t_2 — функция, конструируемая по времени,*

$$\mathsf{TIME}\big(t_1(n)\big) \subsetneq \mathsf{TIME}\big(t_2(n)\big)$$

Следствие 2. *Для любых двух вещественных чисел $1 \leq k < m$:*

$$\mathsf{TIME}(n^k) \subsetneq \mathsf{TIME}(n^m)$$

Следствие 3.

$$\mathsf{P} \subsetneq \mathsf{EXPTIME} = \bigcup_k \mathsf{TIME}(2^{n^k})$$

Полиномиальная иерархия

Взглянем на классы NP и coNP ещё с другого ракурса. Можно сказать, что NP – класс языков, таких, что $\exists\ y \mid A(y) = 1$, где $A(y)$ – проверяемое за полиномиальное время условие (функция, проверяющая принадлежность языку). В данном случае подразумеваем, что y зависит от задачи, как и само условие. Аналогично, coNP – класс языков, таких, что $\forall\ y:\ A(y) = 1$.

Однако, можно определить ещё один класс задач аналогичным образом: Class_1 – класс языков, таких, что $\exists\ y \mid \forall\ z:\ \ A(y,\ z) = 1$ – y и z являются параметрами, зависящими от языка, которые опять же проверяются условием A. Примером задачи из Class_1 является следующая: «Существует ли $\varphi(x)$ – булева функция, такая, что для любого булева вектора $\vec{x}$: $\varphi(\vec{x}) = \psi(\vec{x})$, но $|\varphi(x)| < |\psi(x)|$?», или, проще говоря, «правда ли, что булева функция $\psi(x)$ не минимальна?».

Подобных классов, оказывается, существует довольно-таки много, и для них есть специальные обозначения.

Определение 23. $\sum_k \mathsf{P}$ – *класс задач, таких, что*
$\exists\ x_1 \mid \forall\ x_2\ \exists\ x_3 \mid\ \ldots\ \mathsf{Q}_k x_k : A(x_1,\ x_2,\ \ldots,\ x_k) = 1$, *где* A – *полиномиально проверяемое условие,*
а $\mathsf{Q} \in \{\exists,\ \forall\}$ – *квантор, зависящий от очерёдности:* $\mathsf{Q}_k = \begin{cases} \exists,\ \mathsf{Q}_{k-1} = \forall \\ \forall,\ \mathsf{Q}_{k-1} = \exists \end{cases}$, *причём* $\mathsf{Q}_1 = \exists$.

Определение 24. $\prod_k \mathsf{P}$ – *класс задач, таких, что*
$\forall\ x_1\ \exists\ x_2 \mid \forall x_3\ \ldots\ \mathsf{Q}_k x_k : A(x_1,\ x_2,\ \ldots,\ x_k) = 1$, *где* A – *полиномиально проверяемое условие,*
а $\mathsf{Q} \in \{\exists,\ \forall\}$ – *квантор, зависящий от очерёдности:* $\mathsf{Q}_k = \begin{cases} \exists,\ \mathsf{Q}_{k-1} = \forall \\ \forall,\ \mathsf{Q}_{k-1} = \exists \end{cases}$, *причём* $\mathsf{Q}_1 = \forall$.

Таким образом, $\mathsf{NP} = \sum_1 \mathsf{P}$, а $\mathsf{coNP} = \prod_1 \mathsf{P}$, и если вдруг окажется, что $\mathsf{P} = \mathsf{NP}$, то $\sum_2 \mathsf{P} = \prod_2 \mathsf{P}$, $\sum_3 \mathsf{P} = \prod_3 \mathsf{P}$ и так далее.

Экспоненциальные алгоритмы. Метод локального поиска

Задача о вершинном покрытии

Vertex-Cover

Ввод: $G = (V, E)$, $k \in \mathbb{N}$ – невзвешенный неориентированный граф и число k

Вывод: Ответ на вопрос: "Существует ли вершинное покрытие графа размера $\leqslant k$, то есть $V' \subseteq V \mid |V'| \leqslant k, \forall\ (u, v) \in E : u \in V', v \in V \setminus V'$??"

Скажем сразу: задача $Vertex - Cover \in \mathsf{NPC}$. Вполне естественный вопрос – а как вообще решать подобные задачи в плане реализации, какие использовать методы? Существует три подхода к решению NP-трудных задач:

1. Эффективные алгоритмы для частных случаев – мы наиболее часто сталкиваемся с подобными задачами не в чистом виде, а в качестве упрощённых частных случаев (например, когда граф является деревом), в этом случае можно найти полиномиальный алгоритм-решение без особого труда.

2. Эвристические алгоритмы – для некоторых задач существуют алгоритмы, которые долго работают в худшем случае, но быстро дают ответ в среднем (типичном) случае. Зачастую про такие алгоритмы нет строгих теорем, но на практике оказывается, что работают они вполне неплохо. Разумеется, часто они ищут не точное, а приближённое решение. Но для некоторых задач они ищут и точное решение, в частности, задача коммивояджера для нескольких тысяч городов.

3. Экспоненциальные алгоритмы, отличные от полного перебора – алгоритмы, работающие за экспоненциальное от входа время, но показывающие гораздо лучший результат, нежели полный перебор всех возможных решений на каждом конкретном входе.

Попробуем применить экспоненциальный алгоритм для решения задачи $Vertex - Cover$. Полный перебор подмножеств $V' \subseteq V$ размера k займёт $O(n^k)$ времени, что довольно-таки много. Улучшим результат.

Лемма. *Пусть $(u, v) \in G$ – ребро. Тогда в графе G существует вершинное покрытие размера k если и только если в графе $G \setminus \{u\}$ – граф G без вершины u и всех инцидентных ей рёбер – или в графе $G \setminus \{v\}$ – граф G без вершины v и всех инцидентных ей рёбер – есть вершинное покрытие размера $k - 1$.*

Доказательство. Докажем лемму в обе стороны:

- $\Longrightarrow$

  Пусть существует вершинное покрытие размера k в G. Тогда хотя бы одна вершина ребра (u, v) должна быть в вершинном покрытии. Без ограничения общности, считаем, что u в нём.

  Положим $u \in V'$, где V' – вершинное покрытие графа G, $|V'| = k$. Посмотрим теперь на граф $G \setminus \{u\}$ – фактически при получении этого графа из исходного G мы удаляем из V' вершину u. Так как u покрывает все вершины, не принадлежащие $G \setminus \{u\}$, а в $G \setminus \{u\}$ нет вершин, связанных с u, то значит, $V' \setminus \{u\}$ – множество V' без вершины u – суть вершинное покрытие в $G \setminus \{u\}$. Тогда очевидно, что $|V' \setminus \{u\}| = |V'| - 1 = k - 1$.

- $\Longleftarrow$

  Пусть V' – вершинное покрытие в $G \setminus \{u\}$, $|V'| = k - 1$. Тогда объединим V' с $u \notin G \setminus \{u\}$. Таким образом, мы получили, что $V' \cup \{u\}$ – вершинное покрытие в G, и $|V' \cup \{u\}| = |V'| + 1 = k$.

□

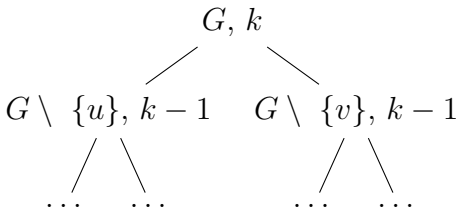
Используя эту лемму, можно сделать эффективный алгоритм:

Алгоритм 7 Vertex-Cover

Ввод: $G = (V, E)$, k

```
1: if |E| = 0 then
2:     return ∅
3: выбираем (u, v) ∈ E
4: if V' = Vertex − Cover(G \ {u}, k − 1) then
5:     return V' ∪ u
6: if V' = Vertex − Cover(G \ {v}, k − 1) then
7:     return V' ∪ v
8: return NULL
```

Получили в итоге дерево рекурсии:



Его высота равна k, всего $\Theta(2^k)$ узлов – число рекурсивных вызовов в худшем случае.

Пускай мы тратим время $O(m)$ на один вызов m полиномиально от количества рёбер. Тогда общее время работы составит $O(m \cdot 2^k)$.

Изначально у нас было время $O(n^k)$, а стало $O(m \cdot 2^k)$, что значительно улучшило время, пускай алгоритм и экспоненциальный! Более того, если $k = \log n$, то $O(m \cdot 2^k) = O(m \cdot n) = O(n^2)$ – полиномиальное от входа время!

Метод локального поиска

Продолжим анализировать задачу $Vertex - Cover$.

Пускай S – какое-то решение (вершинное покрытие). С каждым S ассоциируется его стоимость – $c(S)$. Кроме того, с каждым S ассоциируются «соседние решения» – получаются добавлением или удалением одной вершины из S. Введём несколько обозначений:

Термин	Обозначение	Суть
Решение	S	вершинное покрытие
Стоимость	$c(S)$	$\lvert S\rvert$
Соседние решения	$N(S)$	$S_+ = S \cup \{u\}$ $S_- = S \setminus \{u\}$

Составим схему алгоритма поиска оптимального решения, используя новые термины:

Выбираем S
while выполняется некое условие **do**
    берём соседа S_*, $S_* \in N(S) = \{S_+, S_-\}$
    if S_* лучше S **then**
        $S = S_*$
return лучшее S

Теперь конкретизируем алгоритм – будем использовать **градиентный спуск** для поиска оптимального решения:

Алгоритм 8 Градиентный спуск

1: Выбираем S
2: **while** $\exists\ S_* \in N(S) \mid c(S_*) < c(S)$ **do**
3: $\quad S = \underset{S_* \in N(S)}{\textbf{argmin}}\ c(S_*)$ – берём лучшего из соседей.
4: **return** S

Есть одно НО – градиентный спуск может угодить в локальный минимум, из которого не выберется. Модифицируем градиентный спуск, чтобы он всегда выдавал оптимальные решения. Напрашивается сделать все вершины соседями, но это плохо – получится полный перебор. Сделаем иначе – будем избегать локальных минимумов, иногда «идя вверх». А поможет нам в этом **Алгоритм Метрополиса.**

Алгоритм Метрополиса и Имитация отжига

Немного физики вам в графы:

Определение 25. ***Функцией Гиббса-Больцмана*** *называется следующее чудо:*

$$\exp\left(\frac{-E}{kT}\right),\ \exp(x) = e^x$$

где e –число e, E – энергия, k – постоянная Больцмана, $T > 0$ – температура в Кельвинах.

Данная функция имеет значением величину, пропорциональную вероятности нахождения системы в каком-то состоянии, задаваемом энергией и температурой, в данный момент времени.

Идея функции в том, что энергия и температура системы в различных временных состояниях также разная. И функция Гиббса-Больцмана позволяет нам с определённой вероятностью угадать состояние по его энергии и температуре. Заметим, что если зафиксировать $T = const$, то функция Гиббса-Больцмана станет монотонно убывающей от E.

Алгоритм Метрополиса берёт за основу функцию Гиббса-Больцмана, проводя параллель между вершинным покрытием в графе и состоянием физической системы, фиксируя T и принимая $E(S) = c(S)$, S – состояние (решение).

Алгоритм 9 Metropolis

1: Выбираем S – состояние (решение)
2: **while** $\exists\ S_* \in N(S)$ **do**
3: $\quad$ Выбираем $S_* \in N(S)$
4: $\quad$ **if** $c(S_*) < c(S)$ **then**
5: $\quad\quad S = S_*$
6: $\quad$ **else**
7: $\quad\quad$ с вероятностью $\exp\left(\frac{-\big(c(S_*) - c(S)\big)}{kT}\right)$, заменяем S на S_*
8: **return** S

Пусть $f_S(t)$ – доля тех итераций алгоритма из первых t, в которых мы находились в состоянии S. Тогда в пределе время, проводимое в данном состоянии, пропорционально отношению

величины, пропорциональной вероятности нахождения системы в данном состоянии в момент времени t к сумме всех таких величин для вообще всех состояний:

$$\lim_{t\to\infty} f_S(t) = \frac{\exp\left(\frac{-E(S)}{kT}\right)}{\sum\limits_{S_i}\exp\left(\frac{-E(S_i)}{kT}\right)}$$

Для повышения эффективности применяется также метод **имитации отжига**. Изначально мы делаем T очень большим, а по мере приближения к оптимальному решению (с увеличением номера итерации, то бишь) постепенно уменьшаем T – подобно уменьшению температуры изначально раскалённого металла. Таким образом, изначально наш алгоритм будет метаться от решения к решению, будучи «обожжённым», и по мере «остывания» начнёт предпочитать более корректные решения. При этом, если он попадает в локальный минимум, то с неплохой вероятностью через некоторое время алгоритм оттуда «выпрыгнет», и вечно скакать вокруг оптимума он также не будет, ведь, чем больше итераций, тем больше он остынет, и шансов пропустить нужное решение будет всё меньше.

В итоге, данный алгоритм способен находить вершинное покрытие графа также за экспоненциальное от размера входа время, но при этом работать он будет довольно-таки быстро, в силу превосходства своей эффективности над полиномиальными аналогами.

Метод локального поиска в NP-полных задачах. Алгоритм решения задачи о максимальном разрезе.

Локальный поиск в общем виде

Поговорим подробнее, в каких задачах эффективно применим метод локального поиска. Пусть у нас есть задача, в которой существует множество решений S_1, S_2, ... Перебрать их все за полиномиальное время мы не можем. Цель задачи – найти оптимальное решение. Кроме того, на решениях определено отношение соседства $N(S_i) = \{$множество соседних решений для решения $S_i\}$. Стоит отметить, что перебрать всех соседей мы должны за полиномиальное время, то есть, их не должно быть очень много. Но и очень мало быть не должно, так как иначе действия сведутся к полному перебору.

Для поиска оптимума мы изначально выбираем какое-то одно решение и начинаем последовательно переходить от текущего решения к более хорошему (более близкому к оптимальному) соседу-решению. В итоге мы остановимся на оптимальном по мнению алгоритма решению.

Заметим, что гарантировать результат мы не можем. В частности, алгоритм Градиентного спуска может упасть в локальный минимум, не дойдя до оптимального решения, а алгоритм Метрополиса имеет свойство порой ходить к более худшим соседям.

Рассмотрим подробнее NP-полные задачи, где алгоритм локального поиска может гарантировать результат.

Задача о максимальном разрезе – идея и ассимптотика

Рассмотрим Задачу о максимальном разрезе, но теперь граф взвешенный.

Ввод: $G = (V, E)$ – граф, веса $w(u, v)$, $u, v \in V$. $|V| = n$

Вывод: Разрез в G максимального размера, то есть, такое разбиение вершин графа на два непустых множества $\mathcal{A}$, $\mathcal{B}$, что сумма весов рёбер $w(a, b)$, $a \in \mathcal{A}$, $b \in \mathcal{B}$ максимальна.

Как было доказано ранее, $Max{-}Cut \in \mathsf{NPC}$. Перебрать все разрезы для поиска максимального мы за полином не можем, ибо всего $2^{n-1} - 1$ вариантов – довольно-таки много. Тогда решим задачу методом локального поиска.

Для начала определим отношение соседства. Будем считать разрезы $(\mathcal{A}_i, \mathcal{B}_i)$ и $(\mathcal{A}_j, \mathcal{B}_j)$ соседними, если симметрическая разность множеств $|\mathcal{A}_i \triangle \mathcal{A}_j| = 1$. Иными словами, какая-то одна вершина перешла из одной части разреза в другую, то есть: $\mathcal{A}_j = \mathcal{A}_i \setminus \{v\}$, $\mathcal{B}_j = \mathcal{B}_i \cup \{v\}$. Стоит также отметить, что если у нас есть разрез $(\mathcal{A}, \mathcal{B})$, то у него будет около n соседей – так как мы можем любую вершину графа перекинуть из одной части разреза в другую.

Приведём алгоритм, локальным поиском идущий от начального выбранного разреза к максимальному. Полагаем, что для каждого разреза $(\mathcal{A}, \mathcal{B})$ известно множество $N\big((\mathcal{A}, \mathcal{B})\big)$.

Алгоритм 10 Max-Cut

1: выбрать случайный разрез $(\mathcal{A}, \mathcal{B})$.
2: **repeat**
3: **if** $\exists\, v \in \mathcal{A}$ такая, что $\sum\limits_{u \in \mathcal{A}} w(v, u) > \sum\limits_{u \in \mathcal{B}} w(v, u)$ **then**
4: перемещаем v в $\mathcal{B}$.
5: **if** $\exists\, v \in \mathcal{B}$ такая, что $\sum\limits_{u \in \mathcal{B}} w(v, u) > \sum\limits_{u \in \mathcal{A}} w(v, u)$ **then**
6: перемещаем v в $\mathcal{A}$.
7: **until** не существует ни одной такой вершины v из условий выше
8: **return** получившийся разрез $(\mathcal{A}^*, \mathcal{B}^*)$.

То есть, на каждом шаге мы проверяем условие: если в $\mathcal{A}$ есть такая вершина, для которой сумма весов рёбер, соединяющих её с другими вершинами из $\mathcal{A}$ же больше, чем сумма весов рёбер, соединяющих её с вершинами из другой части разреза, то для увеличения размера разреза мы перемещаем эту вершину в $\mathcal{B}$. Аналогично и для вершины из $\mathcal{B}$. Если таких вершин не осталось, то значит, мы достигли оптимального разреза.

Оценим качество результата, который выдаёт данный алгоритм.

Пусть $(\mathcal{A}^*,\ \mathcal{B}^*)$ – разрез, полученный в результате работы алгоритма **Max-Cut**. Тогда, поскольку алгоритм не смог совершить ещё одну итерацию, придя к такому решению, для каждой вершины данного разреза выполненяются следующие условия:

$$\forall\ v \in \mathcal{A}^* : \sum_{u \in \mathcal{A}^*} w(v,\ u) \leqslant \sum_{u \in \mathcal{B}^*} w(v,\ u)$$
$$\forall\ v \in \mathcal{B}^* : \sum_{u \in \mathcal{B}^*} w(v,\ u) \leqslant \sum_{u \in \mathcal{A}^*} w(v,\ u)$$

Просуммируем теперь соответственно по всем вершинам из $\mathcal{A}$ и из $\mathcal{B}$:

$$\sum_{v \in \mathcal{A}^*} \sum_{u \in \mathcal{A}^*} w(v,\ u) \leqslant \sum_{v \in \mathcal{A}^*} \sum_{u \in \mathcal{B}^*} w(v,\ u)$$
$$\sum_{v \in \mathcal{B}^*} \sum_{u \in \mathcal{B}^*} w(v,\ u) \leqslant \sum_{v \in \mathcal{B}^*} \sum_{u \in \mathcal{A}^*} w(v,\ u)$$

Заметим, что в левых частях неравенств стоит ни что иное, как удвоенная сумма весов рёбер, соединяющих пары вершин из $\mathcal{A}$ и из $\mathcal{B}$ соответсвенно, а в правых частях неравенств стоит ни что иное, как размер разреза $(\mathcal{A}^*,\ \mathcal{B}^*)$:

$$2 \sum_{\substack{v \in \mathcal{A}^* \\ u \in \mathcal{A}^*}} w(v,\ u) \leqslant |(\mathcal{A}^*,\ \mathcal{B}^*)|$$
$$2 \sum_{\substack{v \in \mathcal{B}^* \\ u \in \mathcal{B}^*}} w(v,\ u) \leqslant |(\mathcal{A}^*,\ \mathcal{B}^*)|$$

Сложим оба неравенства и поделим на 2:

$$2 \sum_{\substack{v \in \mathcal{A}^* \\ u \in \mathcal{A}^*}} w(v,\ u) + 2 \sum_{\substack{v \in \mathcal{B}^* \\ u \in \mathcal{B}^*}} w(v,\ u) \leqslant 2|(\mathcal{A}^*,\ \mathcal{B}^*)|$$
$$\Downarrow$$
$$\sum_{\substack{v \in \mathcal{A}^* \\ u \in \mathcal{A}^*}} w(v,\ u) + \sum_{\substack{v \in \mathcal{B}^* \\ u \in \mathcal{B}^*}} w(v,\ u) \leqslant |(\mathcal{A}^*,\ \mathcal{B}^*)|$$

Заметим, что слева у нас стоит сумма всех весов рёбер, соединяющих вершины НЕ из разреза. Прибавим к обеим частям размер разреза, тем самым получив слева сумму весов вообще всех рёбер графа:

$$\sum_{\substack{v \in \mathcal{A}^* \\ u \in \mathcal{A}^*}} w(v,\ u) + \sum_{\substack{v \in \mathcal{B}^* \\ u \in \mathcal{B}^*}} w(v,\ u) \leqslant |(\mathcal{A}^*,\ \mathcal{B}^*)|$$
$$\Downarrow$$
$$\sum_{(v,\ u) \in E} w(v,\ u) \leqslant 2|(\mathcal{A}^*,\ \mathcal{B}^*)|$$
$$\Downarrow$$
$$|(\mathcal{A}^*,\ \mathcal{B}^*)| \geqslant \frac{1}{2} \sum_{(v,\ u) \in E} w(v,\ u)$$

Таким образом, мы получили, что размер разреза не меньше полусуммы весов всех рёбер графа. То есть, алгоритм находит разрез, отличающийся в меньшую сторону от максимального не более, чем в 2 раза, что является неплохим приближённым решением.

Заметим, что если граф не взвешенный, то мы всегда берём и удаляем или прибавляем не просто вершину, а целое ребро. Тогда всего потребуется $O(|E|)$ итераций, что даёт нам полиномиальное от размера графа время работы алгоритма.

Однако, если граф взвешенный, то потребуется $O\left(\sum\limits_{(v,\ u)\in E} w(v,\ u)\right)$ запусков алгортима, чтобы получить оптимум – алгоритм будет псевдополиномиальным, так как будет выполнять полином действий на каждом шаге, но в итоге получится не полиномиальным от входа. Попробуем немного ослабить точность алгоритма, чтобы получить полиномиальный алгоритм.

Пусть имеется некая константа $\varepsilon > 0$. Будем выбирать вершину для перекидывания только в том случае, если перебрасывание данной вершины из одной части разреза в другую увеличивает размер разреза хотя бы в $\left(1 + \dfrac{2\varepsilon}{n}\right)$ раз, то есть, если мы перешли от решения $(\mathcal{A}_1,\ \mathcal{B}_1)$ к решению $(\mathcal{A}_2,\ \mathcal{B}_2)$, то выполнено:

$$\frac{|(\mathcal{A}_2,\ \mathcal{B}_2)|}{|(\mathcal{A}_1,\ \mathcal{B}_1)|} \geqslant 1 + \frac{2\varepsilon}{n}$$

Теперь старые условия для полученного разреза $(\mathcal{A}^*,\ \mathcal{B}^*)$ работать уже не будут. Перепишем их для нового алгоритма:

$$\forall\ v \in \mathcal{A}^* : \sum_{u\in\mathcal{A}^*} w(v,\ u) \leqslant \sum_{u\in\mathcal{B}^*} w(v,\ u) + \frac{2\varepsilon}{n}|(\mathcal{A}^*,\ \mathcal{B}^*)|$$
$$\forall\ v \in \mathcal{B}^* : \sum_{u\in\mathcal{B}^*} w(v,\ u) \leqslant \sum_{u\in\mathcal{A}^*} w(v,\ u) + \frac{2\varepsilon}{n}|(\mathcal{A}^*,\ \mathcal{B}^*)|$$

Опять же суммируем по всем v:

$$\sum_{v\in\mathcal{A}^*}\sum_{u\in\mathcal{A}^*} w(v,\ u) \leqslant \sum_{v\in\mathcal{A}^*}\left(\sum_{u\in\mathcal{B}^*} w(v,\ u) + \frac{2\varepsilon}{n}|(\mathcal{A}^*,\ \mathcal{B}^*)|\right)$$
$$\sum_{v\in\mathcal{B}^*}\sum_{u\in\mathcal{B}^*} w(v,\ u) \leqslant \sum_{v\in\mathcal{B}^*}\left(\sum_{u\in\mathcal{A}^*} w(v,\ u) + \frac{2\varepsilon}{n}|(\mathcal{A}^*,\ \mathcal{B}^*)|\right)$$
$$\Downarrow$$
$$2\sum_{\substack{v\in\mathcal{A}^*\\ u\in\mathcal{A}^*}} w(v,\ u) \leqslant |(\mathcal{A}^*,\ \mathcal{B}^*)| + |\mathcal{A}^*|\cdot\frac{2\varepsilon}{n}|(\mathcal{A}^*,\ \mathcal{B}^*)|$$
$$2\sum_{\substack{v\in\mathcal{B}^*\\ u\in\mathcal{B}^*}} w(v,\ u) \leqslant |(\mathcal{A}^*,\ \mathcal{B}^*)| + |\mathcal{B}^*|\cdot\frac{2\varepsilon}{n}|(\mathcal{A}^*,\ \mathcal{B}^*)|$$

Сложим оба неравенства и поделим на 2:

$$2\sum_{\substack{v\in\mathcal{A}^*\\ u\in\mathcal{A}^*}} w(v,\ u) + 2\sum_{\substack{v\in\mathcal{B}^*\\ u\in\mathcal{B}^*}} w(v,\ u) \leqslant 2|(\mathcal{A}^*,\ \mathcal{B}^*)| + \underbrace{(|\mathcal{A}^*| + |\mathcal{B}^*|)}_{=|V|=n}\cdot\frac{2\varepsilon}{n}|(\mathcal{A}^*,\ \mathcal{B}^*)|$$
$$\Downarrow$$
$$\sum_{\substack{v\in\mathcal{A}^*\\ u\in\mathcal{A}^*}} w(v,\ u) + \sum_{\substack{v\in\mathcal{B}^*\\ u\in\mathcal{B}^*}} w(v,\ u) \leqslant |(\mathcal{A}^*,\ \mathcal{B}^*)| + \varepsilon\cdot|(\mathcal{A}^*,\ \mathcal{B}^*)|$$

Прибавим к обеим частям размер разреза, тем самым получив слева сумму весов вообще всех рёбер графа:

$$\sum_{(v,\ u)\in E} w(v,\ u) \leqslant 2|(\mathcal{A}^*,\ \mathcal{B}^*)| + \varepsilon \cdot |(\mathcal{A}^*,\ \mathcal{B}^*)| = (2+\varepsilon)\cdot|(\mathcal{A}^*,\ \mathcal{B}^*)|$$

$$\Downarrow$$

$$|(\mathcal{A}^*,\ \mathcal{B}^*)| \geqslant \frac{1}{2+\varepsilon} \sum_{(v,\ u)\in E} w(v,\ u)$$

Как можно заметить, мы не особо потеряли в точности. Алгоритм опять же работает за полином. Однако, неизвестно, сколько потребуется запусков – вдруг опять выйдет долго? Проверим время работы.

При перемещении вершины из $\mathcal{A}$ в $\mathcal{B}$ или из $\mathcal{B}$ в $\mathcal{A}$ размер $(\mathcal{A},\ \mathcal{B})$ увеличивается хотя бы в $\left(1+\dfrac{2\varepsilon}{n}\right)$ раз. Тогда по прошествии $\dfrac{n}{\varepsilon}$ итераций размер разреза увеличится хотя бы в 2 раза:

$$\left(1+\frac{2\varepsilon}{n}\right)^{\frac{n}{\varepsilon}} \geqslant \{\text{неравенство Бернулли}\} \geqslant 1+\left(\frac{2\varepsilon}{n}\right)\cdot\frac{n}{\varepsilon} = 1+2 \geqslant 2$$

Тогда всего итераций, в силу оценки на размер получаемого разреза по отношению к максимальному, потребуется

$$\frac{n}{\varepsilon}\log_2 \sum_{(v,\ u)\in E} w(v,\ u)$$

итераций – теперь алгоритм уже полиномиален (и даже лучше) от размера входа! Таким образом, мы получили требуемый приближённый алгоритм поиска максимального разреза во взвешенном графе. Однако, мы действовали с предположением, что соседние решения уже построены и заранее, и для каждого разреза $(\mathcal{A},\ \mathcal{B})$ известно множество $N\big((\mathcal{A},\ \mathcal{B})\big)$. А ведь ещё неизвестно, сколько времени мы будем подсчитывать соседние решения.

Задача о максимальном разрезе – эвристика

Покажем, что построение множества $N\big((\mathcal{A},\ \mathcal{B})\big)$ для разреза $(\mathcal{A},\ \mathcal{B})$ происходит за полиномиальное время. Приведём алгоритм, находящий самих соседей.

Алгоритм 11 Calculate-Neighbours$\big((\mathcal{A},\ \mathcal{B})\big)$

зафиксируем начальный разрез $(\mathcal{A}_0,\ \mathcal{B}_0) = (\mathcal{A},\ \mathcal{B})$.
$C = V$ – множество вершин-кандидатов для перекидывания
for $i = 1$ **to** n **do**
    for $u \in C$ **do**
        перемещаем u в $(\mathcal{A}_{i-1},\ \mathcal{B}_{i-1})$, получив $(\mathcal{A}',\ \mathcal{B}')$
        считаем $|((\mathcal{A}',\ \mathcal{B}'))|$, зафиксировав результат для этой вершины u
        возвращаем u в C
    выбираем u с наилучшим результатом – u_{best}
    считаем $(\mathcal{A}_i,\ \mathcal{B}_i)$, полученный из $(\mathcal{A}_{i-1},\ \mathcal{B}_{i-1})$ перемещением вершины u_{best}, и фиксируем
    $C = C \setminus \{u_{best}\}$
return $\big\{(\mathcal{A}_1,\ \mathcal{B}_1),\ (\mathcal{A}_2,\ \mathcal{B}_2),\ \ldots,\ (\mathcal{A}_n,\ \mathcal{B}_n)\big\}$

В итоге мы получим n разрезов, из которых уже выбираем лучший.

Задача сегментации и Многотерминальный разрез

Постановка задачи

Представим, что у нас есть изображение со множеством объектов: люди, деревья, звери, машины и подобное. Мы хотим разбить пиксели изображения на множества так, чтобы два пикселя относились к одному и тому же множеству если и только если они оба принадлежат одному и тому же объекту (сегменту изображения). Данную задачу можно проинтерпретировать на языке графов, положив, что каждому пикселю изображения соответствует своя вершина. Тогда рассмотрим следующую **Задачу сегментации**:

Segmentation

Ввод: $G = (V, E)$ – неориентированный граф
L – метки сегментов, $|L| = l$
$P = (p_{uv})$ – матрица рёбер $(u, v) \in E$, соединяющих вершины-пиксели. Мы платим штраф p_{uv}, если метки у вершин u и v различны.
множество $\{c_p(a)\}$ – штраф за отнесение пикселя $p \in V$ к метке $a \in L$.

Вывод: классификация – разбиение на множества, обозначенные различными метками – $\kappa : V \to L$, такое, что сумма $\sum\limits_{p \in V} c_p\big(\kappa(p)\big) + \sum\limits_{\substack{u,\ v \in V \\ \kappa(u) \neq \kappa(v)}} p_{uv}$ минимальна.

Заметим, что p_{uv} и $c_p(a)$ могут быть разных размеров, поэтому для разных задач сегментации соотношение величин и сами классификации могут существенно отличаться.

Будем оценивать сложность данной задачи. Но сначала поговорим об ещё одной.

Задача о многотерминальном разрезе

Ранее в курсе алгоритмов мы говорили о **Задаче о минимальном разрезе**, в которой фигурировали две **терминальные** вершины s и t. Она лежит в классе P, так как двойственной к ней является **Задача о максимальном потоке**, для которой существуют алгоритмы решения, работающие за полиномиальное от размера графа время. Теперь же рассмотрим ту же задачу, но для нескольких терминальных вершин T, $|T| \geqslant 3$. Называется она **Задачей о многотерминальном разрезе**.

Multiterminal-Cut

Ввод: $G = (V, E)$ – неориентированный граф, $w(u, v) \geqslant 0$, $(u, v) \in E$ – веса рёбер, $T \subseteq V$ – подмножество терминальных вершин, число k

Вывод: Ответ на вопрос: "Существует ли такое подмножество рёбер $C \subseteq E$, такое, что в графе $G(C) = (V, E \setminus C)$ все вершины $t \in T$ в разных компонентах связности, причём C минимально по вложению и $\sum\limits_{(u,\ v) \in C} w(u, v) \leqslant k$ (сумма весов рёбер разреза минимальна)??"

Иными словами, мы «разрезаем» граф на компоненты связности, и минимизируем суммарный вес рёбер, соединяющих вершины из разных компонент (из разных частей разреза). Такие рёбра будет называть *удалёнными* рёбрами.

Данная задача поможет нам оценить сложность Задачи сегментации. Однако, для начала необходимо узнать, к какому классу языков принадлежит сама $Multiterminal - Cut$. В этом нам поможет старая добрая **Задача о максимальном разрезе**:

Max-Cut

Ввод: $G = (V, E)$, $k \in \mathbb{N}$ – невзвешенный неориентированный граф

Вывод: Ответ на вопрос: "Существует ли (A, B) – разбиение вершиин графа на два множества, такое, что $A \cup B = V$, $A \cap B = \varnothing$, $A \neq \varnothing$, $B \neq \varnothing$, причём количество рёбер, соединяющих вершины из разных множеств – $|\{(u, v) \mid u \in A, v \in B\}| \geqslant k$??"

Теорема 6. $Multiterminal - Cut \in \mathsf{NPC}$

Доказательство. Заметим, что для данной теоремы достаточно рассмотреть случай с $|T| = 3$, так как для $|T| > 3$ задача пропорциональная данному случаю, и если уже для трёх компонент $Multiterminal - Cut$ NP-полна, то для большего числа оных и подавно.

Для начала докажем следующую лемму:

Лемма. $Max - Cut \leqslant_p Multiterminal - Cut$

Доказательство. Приведём функцию f, вычислимую за полиномиальное время, которая сводит задачу $Max - Cut$ к задаче о поиске многотерминального разреза, строя необходимый граф по входным данным.

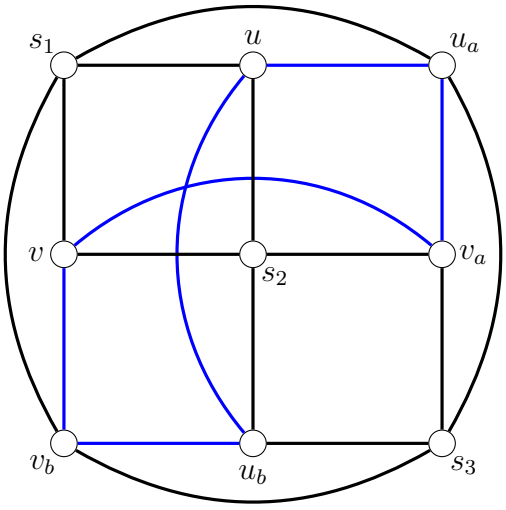
Имеем граф $G = (V, E)$. Построим по нему граф F для $Multiterminal - Cut$. Зададим множество вершин данного графа так:

- 3 вершины s_1, s_2, s_3 – они будут терминальными
- V – множество вершин исходного графа G
- Для каждого ребра $(u, v) \in E$ сделаем по 4 вершины: u_a, u_b, v_a, v_b, получив тем самым $4|E|$ вершин к общему числу оных.

Зафиксируем для графа плоскость, на которой он «нарисован», на плоскости зададим 2 оси – *вертикальную* и *горизонтальную*. Будем считать, что любые n вершин в нашем графе можно поперетаскивать (на плоскости) так, чтобы за неизменением положения остальных вершин эти n выбранных лежали на одной прямой (абстрактной), причём как горизонтальной (параллельной горизонтальной оси плоскости) или вертикальной (параллельной вертикальной оси плоскости), так и диагональной (договоримся, что диагональные прямые расположены под углом $-45°$ к горизонтали). Конструкции графа это не меняет, изменится только визуальное восприятие. Необходимо это лишь для упрощения используемой терминологии и улучшения понятности действий. Назовём такой процесс перетаскивания вершин **выравниванием**.

Теперь для каждого каждого ребра $(u, v) \in E$ строим следующую конструкцию. Выравним вершины s_1 и u так, чтобы они лежали на одной горизонтали. Выравним вершины s_1 и v, u и s_2 так, чтобы они лежали на одних вертикалях соответственно. Выравним вершины s_1, s_2, s_3 так, чтобы они лежали на одной диагонали. Для «дополнительных» вершин u_a, u_b, v_a, v_b ребра (u, v) проделаем следующее: выравним вершины s_1, u, u_a по горизонтали, s_1, v, v_b по вертикали. Далее, выравним вершины u, s_2, u_b по вертикали, вершины v, s_2, v_a по горизонтали. После чего выравним вершины u_a, v_a, s_3 и v_b, u_b, s_3 по вертикали и горизонтали соответственно.

Заведём два типа рёбер – **чёрные** веса 4 и **синие** веса 1. Проводим рёбра между вершинами, и в итоге получаем такой объект:



Назовём его **гаджет--ребром**.

Получается он так: соединяем нетерминальные вершины (u, v, u_a, u_b, v_a, v_b) **чёрными** рёбрами с терминальными вершинами (s_1, s_2, s_3), причём с теми и только теми, что выравнены с данными на одной горизонтали или вертикали. Соединяем нетерминальные вершины u и v **синими** рёбрами с теми и только теми нетерминальными вершинами, что выравнены с ними на одной горизонтали или вертикали (то есть u и v с вершинами u_a, u_b и v_a, v_b соответственно, и дополнительные вершины между собой).

Строим по гаджет-ребру для каждого ребра $(u, v) \in E$, причём вершины s_1, s_2, s_3 используются одни и те же (в процессе выравниваем граф, как нам удобно для построения). В итоге мы получаем построенный граф F.

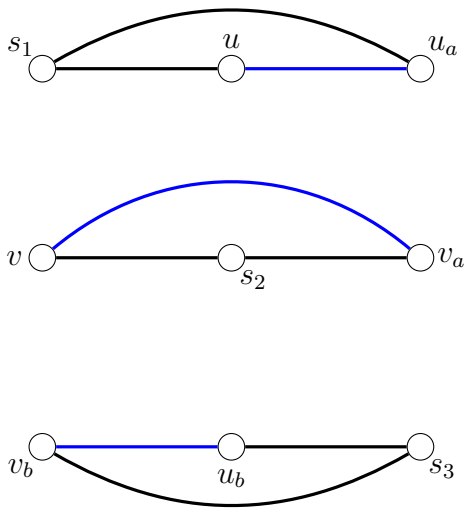
Ясно, что построение графа занимает полиномиальное от размера входных данных (то есть изначального графа $G = (V, E)$) время – мы создаём $\Theta(E)$ новых вершин и проводим константное число рёбер при построении каждого **гаджет--ребра**, получая опять $\Theta(E)$ времени. Выравнивания на время не влияют никак.

Опишем подробнее, как будет выглядеть многотерминальный (точнее, 3-терминальный) разрез в F после того, как функция f построила данный граф по G. Действуем так: создаём 3 множества вершин – U_1, U_2, U_3. Пусть в G существует разрез (V_1, V_2). Он уже «порезал» что-то из F, разбив некоторые вершины графа на две компоненты. Сохраним разбиение, поместив V_1 в U_1, V_2 в U_2. После этого определяем s_1, s_2, s_3 в множества U_1, U_2, U_3 соответственно.

Осталось разбить дополнительные $4|E|$ вершин по трём компонентам. Пускай у нас имеется ребро $(u, v) \in E$. Возьмём соответствующее данному ребру **гаджет--ребро**, и рассмотрим 4 различных случая, определяемые множествами вершин u и v при разбиении в графе F.

1. $u \in U_1$, $v \in U_2$

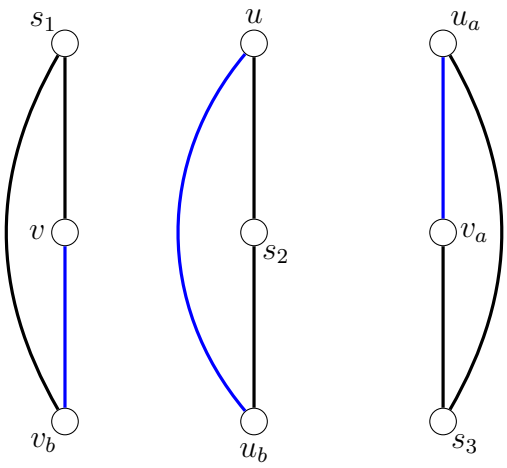
   $u \in U_1$, значит, u и s_1 в одной компоненте. $v \in U_2$, значит, v и s_2 в одной компоненте. При этом они все в разных с s_3 компонентах. Отсюда следует, что у нас имеется возможность «порезать» **гаджет--ребро** по горизонталям, на которых выравнены соответственно u и s_1, v и s_2 и s_3. Мы пользуемся этой возможностью (так как подобный разрез нас устраивает), и оно примет следующий вид (рёбра разреза мы удаляем):



   Все вертикальные рёбра будут в разрезе. Удалено 6 **чёрных** рёбер и 3 **синих**.

2. $u \in U_2$, $v \in U_1$

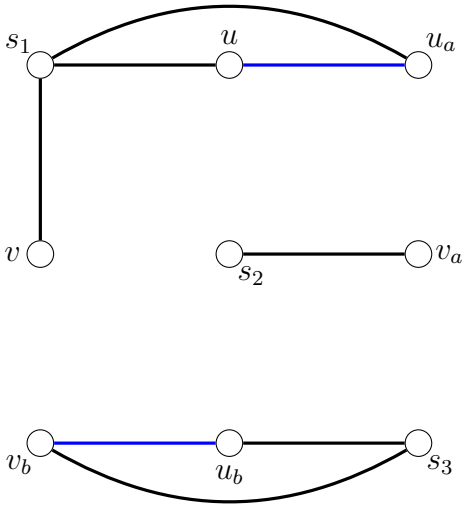
   $u \in U_2$, значит, u и s_2 в одной компоненте. $v \in U_1$, значит, v и s_1 в одной компоненте. При этом они все в разных с s_3 компонентах. Отсюда следует, что у нас имеется возможность «порезать» **гаджет--ребро** по вертикалям, на которых выравнены соответственно u и s_2, v и s_1 и s_3, и оно примет следующий вид (рёбра разреза мы удаляем):



Все горизонтальные рёбра будут в разрезе. Удалено 6 **чёрных** рёбер и 3 **синих**.

3. $u \in U_1$, $v \in U_1$

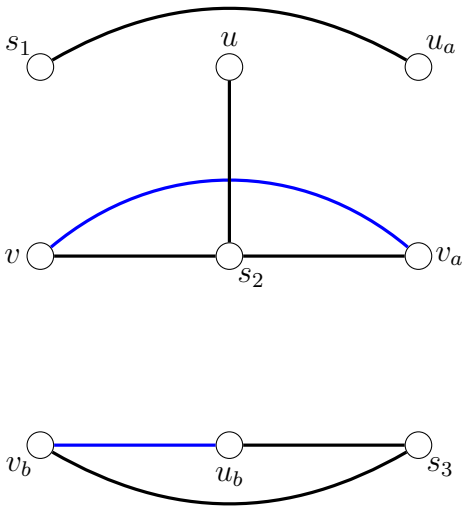
   $u \in U_1$, $v \in U_1$, значит, u, v и s_1 будут в одной компоненте. При этом они, s_2 и s_3 все лежат в разных компонентах. Отсюда следует, что наше **гаджет--ребро** «порежется» так: в компоненту U_1 войдут вершины, выравненные на одной горизонтали с s_1, а также вершина v. В компоненту U_3 войдут вершины, выравненные на одной горизонтали с s_3. Оставшиеся вершины и s_2 пойдут в U_2. В итоге **гаджет--ребро** примет следующий вид (рёбра разреза мы удаляем):



   В разрезе будут все вертикальные рёбра, за исключением ребра $(s_1,\ v)$. Вместо него в разрез идут ребра $(s_2,\ v)$ и $(v,\ v_a)$ – из-за другой компоненты вершины v. Удалено 6 **чёрных** рёбер и 4 **синих**.

4. $u \in U_2$, $v \in U_2$

   $u \in U_2$, $v \in U_2$, значит, u, v и s_2 будут в одной компоненте. При этом они, s_1 и s_3 все лежат в разных компонентах. Отсюда следует, что наше **гаджет--ребро** «порежется» так: в компоненту U_2 войдут вершины, выравненные на одной горизонтали с s_2, а также вершина u. В компоненту U_3 войдут вершины, выравненные на одной горизонтали с s_3. Оставшиеся вершины и s_1 пойдут в U_1. В итоге **гаджет--ребро** примет следующий вид (рёбра разреза мы удаляем):



В разрезе будут все вертикальные рёбра, за исключением ребра $(s_2,\ u)$. Вместо него в разрез идут ребра $(s_1,\ u)$ и $(u,\ u_a)$ – из-за другой компоненты вершины u. Удалено 6 **чёрных** рёбер и 4 **синих**.

Таким образом, для каждого ребра $(u,\ v) \in E$ исходного графа соответствующее данному ребру **гаджет--ребро** графа F будет побито на компоненты связности одним из 4-х вышеуказанных случаев, в зависимости от исходного разбиения вершин u и v.

Покажем теперь, что сведение корректно, то есть, что размер максимального разреза k в исходном графе G соответствует определённому размеру минимального многотерминального разреза в получившемся графе F, зависящему от размера максимального разреза G (то есть, решение одной задачи аналогично решению другой задачи, и решения взаимосвязаны). Иными словами, $\varpi \in Max - Cut \iff f(\varpi) \in Multiterminal - Cut$. Однако, мы сделаем нечто помощнее. Будем доказывать более сильное утверждение – у задачи $\varpi \in Max - Cut$ существует решение (разрез) размера $\geqslant k$ если и только если для задачи $f(\varpi) \in Multiterminal - Cut$ существует решение (разрез) резмера $\leqslant 28|E| - k$:

- $\exists\ \geqslant k \to \varpi \in Max - Cut \Longrightarrow \exists\ \leqslant 28|E| - k \to f(\varpi) \in Multiterminal - Cut$

  Посчитаем суммарный вес удаляемых рёбер (то есть рёбер разреза) в каждом из 4-х возможных случаев разбиения **гаджет--ребра** в графе F, отвечающего ребру $(u,\ v)$ исходного графа. Случаи 1 и 2 дают 6 **чёрных** рёбер и 3 **синих**, суммарный вес которых равен $6 \cdot 4 + 3 \cdot 1 = 27$. Случаи 3 и 4 дают 6 **чёрных** рёбер и 4 **синих**, суммарный вес которых равен $6 \cdot 4 + 4 \cdot 1 = 28$. Получается, что если вершины u и v лежат в разных компонентах в исходном графе, то они дают вклад 27 в стоимость многотерминального разреза графа F, а если в одной компоненте, то вклад равен 28. Значит:

  $$\text{вклад в одной компоненте} = \text{вклад в разных компонентах} + 1$$

  Следовательно, чтобы посчитать суммарный размер многотерминального разреза F, можно сначала посчитать суммарный вклад вообще всех рёбер, будто их вершины лежат в одинаковых компонентах, а потом вычесть количество рёбер, вершины которых лежат в разных компонентах – убираем по единичке вклада с каждого такого ребра, чтобы получить нужный размер. Заметим, что количество таких рёбер, вершины которых лежат в разных компонентах разбиения $(U_1,\ U_2,\ U_3)$, в точности равно размеру разреза $|(V_1,\ V_2)| = k^*$ исходного графа G. Отсюда:

  $$|(U_1,\ U_2,\ U_3)| = 28|E| - |(V_1,\ V_2)| = 28|E| - k^* \leqslant 28|E| - k$$

  То есть, в F существует минимальный многотерминальный разрез необходимого размера: $\exists\ \leqslant 28|E| - k \to f(\varpi) \in Multiterminal - Cut$.

- $\exists\ \leqslant 28|E| - k \to f(\varpi) \in Multiterminal - Cut \Longrightarrow \exists\ \geqslant k \to \varpi \in Max - Cut$

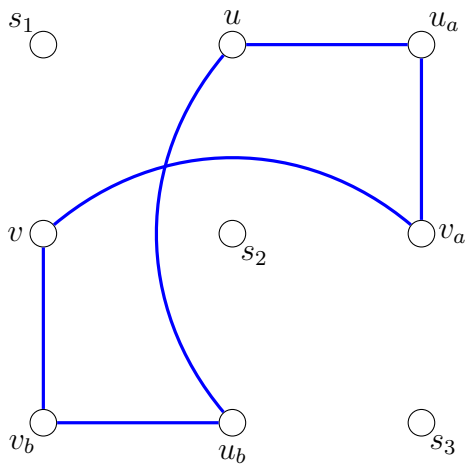
  Пускай в F существует разез (U_1, U_2, U_3) размера $28|E| - k^* \leqslant 28|E| - k$.

  Рассмотрим **гаджет--ребро**, отвечающее ребру (u, v). Покажем, что данное ребро даёт вклад $\leqslant 27$ в стоимость минимального многотерминального разреза если и только если вершины u и v находятся в разных компонентах связности, в противном же случае – u и v в одной компоненте – они дадут вклад $\geqslant 28$.

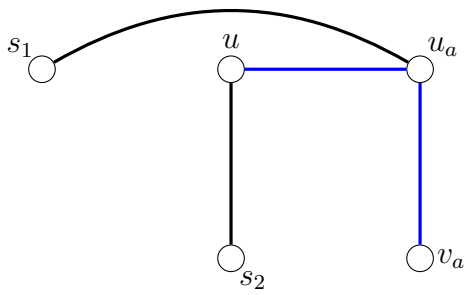
  Пускай у нас имеется **гаджет--ребро**, дающее вклад $\leqslant 27$. По построению, каждая нетерминальная вершина соединена **чёрными** рёбрами с терминальными вершинами, выравненными с данной на одних и тех же осях. Возьмём какую-то нетерминальную вершину p данного **гаджет--ребра**. Заметим, что из каждой пары **чёрных** рёбер (s_1, p) и (s_2, p) хотя бы одно будет удалено (войдёт в разрез) – иначе существовал бы путь $s_1 \to p \to s_2$, что в силу терминальности вершин s_1 и s_2 невозможно. Всего таких **чёрных** рёбер в каждом из 4-х случаев разрезания **гаджет--ребра** по 6 штук, значит, вклад данного **гаджет--ребра** в размер разреза (U_1, U_2, U_3) уже составляет как минимум $6 \cdot 4 = 24$.

  Заметим, что больше мы не можем удалить ни одного **чёрного** рёбра – вклад тогда станет как равен как минимум 28, в то время как суммарный вклад рассматриваемого **гаджет--ребра** $\leqslant 27$. Значит, из каждой пары рёбер вида (s_i, p), (s_j, p), соединяющих данную нетерминальную вершину p с терминальными, останется ровно одно ребро.

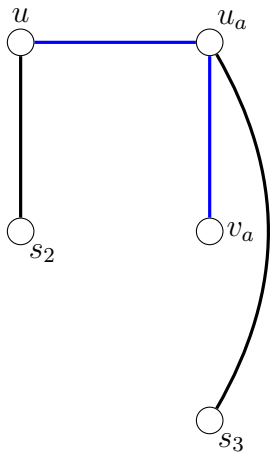
  Теперь рассмотрим **синие** рёбра данного **гаджет--ребра**. Заметим, что они образуют цикл:



  Рассмотрим правый верхний уголок. Пускай мы оставили два смежных **синих** рёбра – (u_a, u) и (u_a, v_a). Для вершины u из двух **чёрных** рёбер осталось ровно одно – пускай это будет (s_2, u). Также, для вершины u_a из двух **чёрных** рёбер осталось ровно одно – пускай это будет (s_1, u_a).

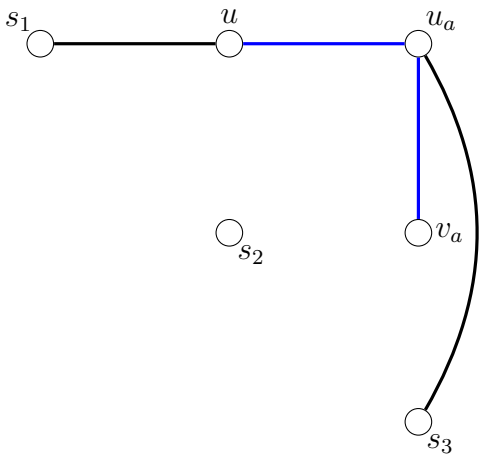


Но тогда существует путь $s_1 \to u_a \to u \to s_2$, что невозможно в силу терминальности вершин s_1 и s_2. Следовательно, ребро (s_1, u_a) остаться не могло. Пускай осталось (s_3, u_a).

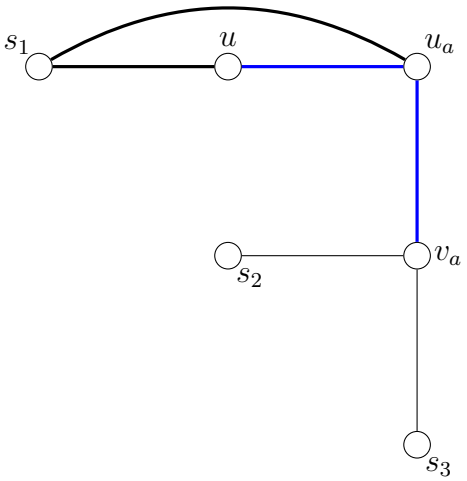


Но тогда, вне зависимости от рёбер между вершинами s_2, s_3 и v_a существует путь $s_2 \to u \to u_a \to s_3$, что невозможно в силу терминальности вершин s_1 и s_3.

Обратимся теперь к другому случаю: для вершины u осталось ребро (s_1, u). Тогда для вершины u_a из двух **чёрных** рёбер осталось ровно одно – пускай это будет (s_3, u_a).

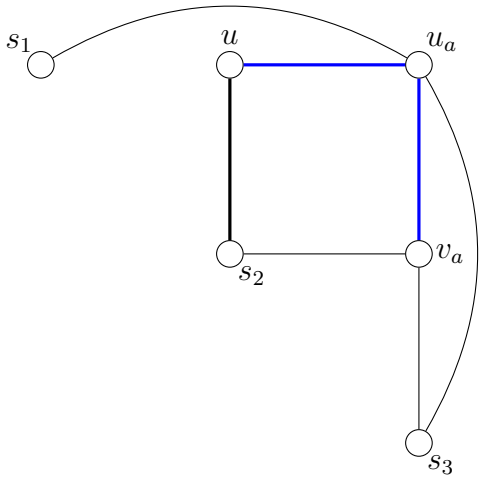


Но тогда, вне зависимости от рёбер между вершинами s_2, s_3 и v_a существует путь $s_1 \to u \to u_a \to s_3$, что невозможно в силу терминальности вершин s_1 и s_3. Следовательно, ребро (s_3, u_a) остаться не могло. Пускай тогда осталось ребро (s_1, u_a).



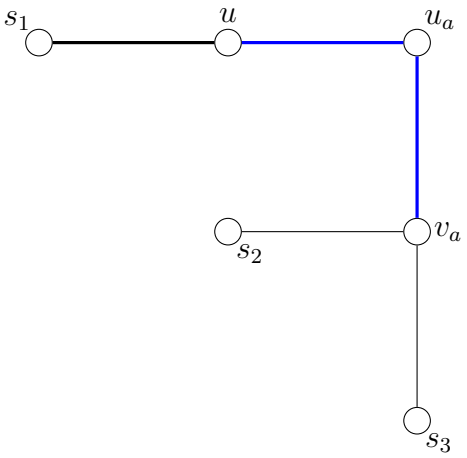
Для вершины v_a из двух **чёрных** рёбер осталось ровно одно. Пускай это будет ребро $(s_2,\ v_a)$. Но тогда существует путь $s_1 \to u_a \to v_a \to s_2$, что невозможно в силу терминальности вершин s_1 и s_2. Пускай осталось ребро $(s_3,\ v_a)$. Но тогда существует путь $s_1 \to u_a \to v_a \to s_3$, что невозможно в силу терминальности вершин s_1 и s_3.

Теперь рассмотрим подробнее вершину v_a. Для вершины u из двух **чёрных** рёбер осталось ровно одно – пускай это будет $(s_2,\ u)$.



Для вершины u_a из двух **чёрных** рёбер осталось ровно одно. Пускай это будет ребро $(s_1,\ u_a)$. Но тогда существует путь $s_1 \to u_a \to u \to s_2$, что невозможно в силу терминальности вершин s_1 и s_2. Пускай осталось ребро $(s_3,\ u_a)$. Но тогда существует путь $s_2 \to u \to u_a \to s_3$, что невозможно в силу терминальности вершин s_2 и s_3.

Обратимся теперь к другому случаю: для вершины u ребро $(s_1,\ u)$.



Для вершины v_a из двух **чёрных** рёбер осталось ровно одно. Пускай это будет ребро $(s_2,\ v_a)$. Но тогда существует путь $s_1 \to u \to u_a \to v_a \to s_2$, что невозможно в силу терминальности вершин s_1 и s_2. Пускай осталось ребро $(s_3,\ v_a)$. Но тогда существует путь $s_1 \to u \to u_a \to v_a \to s_3$, что невозможно в силу терминальности вершин s_1 и s_3.

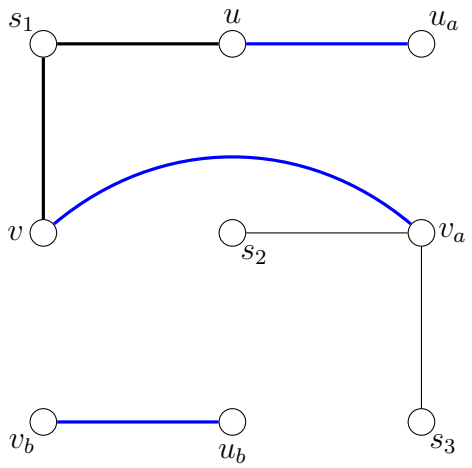
По итогу, во всех возможных случаях разрезания **гаджет--ребра** при условии, что в правом уголке мы оставили два смежных **синих** рёбра – $(u_a,\ u)$ и $(u_a,\ v_a)$, мы пришли к противоречию. Значит, двух смежных **синих** рёбер для вершины u_a правого верхнего уголка быть не может, следовательно, останется только какое-то одно из них – второе будет удалено. Аналогично для левого нижнего уголка – для вершины v_b не может быть двух смежных **синих** рёбер, и одно из них также попадёт в разрез.

Получается, что с каждого уголка мы должны удалить по одному **синему** ребру. Кроме того, в силу специфики 4-х возможных случаев разбиения вершин **гаджет--ребра**, мы вынуждены удалить одно из рёбер (u, u_b) или (v, v_a) – оба они остаться не могут, так как вершины u и v не являются как-либо выравненными. Следовательно, хотя бы 3 **синих** ребра. Мы должны удалить, откуда суммарный вклад **гаджет--ребра** становится равен как минимум $3 + 24 = 27$.

Посмотрим на наши 4 случая разбиения **гаджет--ребра**. Вышеуказанная ситуация возможна если и только если было удалено как минимум 3 **синих** ребра, и не более 3-х **синих** рёбер осталось в графе.

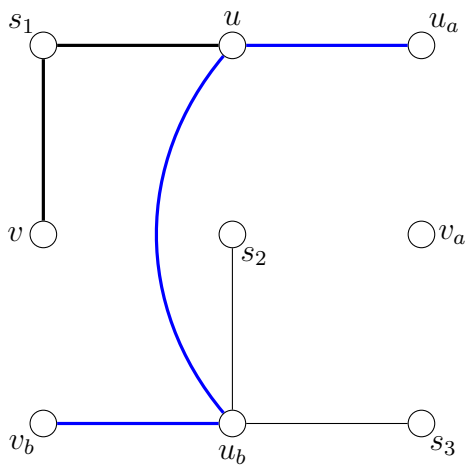
Если вершины u, v находятся в разных компонентах, тогда вклад **гаджет--ребра** в точности равен $27 \leqslant 27$ – условие выполнено. Пусть теперь u и v при разбиении попали в одну компоненту. Положим, что $u \in U_1$, $v \in U_1$. Покажем, что в этом случае ещё хотя бы одно **синее** ребро попало в разрез. Пускай остались 3 **синих** ребра.

Положим, осталось (v, v_a). Для вершины v_a из двух **чёрных** рёбер осталось ровно одно. Пусть это будет ребро (s_2, v_a). Тогда существует путь $s_1 \to v \to v_a \to s_2$, что невозможно в силу терминальности вершин s_1 и s_2. Пусть осталось ребро (s_3, v_a). Тогда существует путь $s_1 \to v \to v_a \to s_3$, что невозможно в силу терминальности вершин s_1 и s_3.



Значит, **синее** ребро (v, v_a) также попадёт в разрез.

Положим, осталось (u, u_b). Для вершины u_b из двух **чёрных** рёбер осталось ровно одно. Пусть это будет ребро (s_2, u_b). Тогда существует путь $s_1 \to u \to u_b \to s_2$, что невозможно в силу терминальности вершин s_1 и s_2. Пусть осталось ребро (s_3, u_b). Тогда существует путь $s_1 \to u \to u_b \to s_3$, что невозможно в силу терминальности вершин s_1 и s_3.



Значит, **синее** ребро $(u,\, u_b)$ также попадёт в разрез.

Аналогично для случая, когда $u \in U_2$, $v \in U_2$ – ещё хотя бы одно **синее** ребро будет удалено.

По итогу, мы получаем, что если вершины u и v находятся в одной компоненте связности, то ещё хотя бы одно **синее** ребро попало в разрез. То есть, вклад **гаджет--ребра** равен хотя бы $1 + 3 + 24 = 28 \geqslant 28$ – условие также выполнено.

Из выполнения данного условия следует, что ребро $(u,\, v)$ даёт вклад 27 в размер минимального многотерминального разреза если и только если вершины u и v находятся в разных компонентах связности в графе F. Значит, поскольку мы из суммы вкладов всех рёбер при одинаковых компонентах убираем по единичке с вклада каждого такого ребра, чтобы получился нужный размер, и итоговых размер $|(U_1,\, U_2,\, U_3)| = 28 - k^*$, то число k^* в точности равно числу рёбер $(u,\, v)$ графа F таких, что $u \in U_1$ и $v \in U_2$. По построению, каждому такому ребру $(u,\, v)$ графа F соответствует аналогичное ребро $(u,\, v)$ в G с сохранением разбиения на компоненты. А так как $V_1 \subseteq U_1$, $V_2 \subseteq U_2$, то значит, это число k^* также в точности равно числу рёбер $(u,\, v) \in E$ таких, что $u \in V_1$, $v \in V_2$. А это ни что иное, как размер разреза $(V_1,\, V_2)$ графа G. Ну а поскольку

$$|(U_1,\, U_2,\, U_3)| = 28|E| - k^* \leqslant 28|E| - k \Longrightarrow |(V_1,\, V_2)| = k^* \geqslant k$$

то значит, наш разрез $(V_1,\, V_2)$ как раз таки нужного размера, и $\exists\ \geqslant k \to \varpi \in Max - Cut$.

Таким образом, мы привели функцию f, вычислимую за полиномиальное время, такую, что выполнено сильное условие: у задачи $\varpi \in Max - Cut$ существует решение (разрез) размера $\geqslant k$ если и только если для задачи $f(\varpi) \in Multiterminal - Cut$ существует решение (разрез) резмера $\leqslant 28|E| - k$.

Значит, $\exists\ \geqslant k \to \varpi \in Max - Cut \iff \exists\ \leqslant 28|E| - k \to f(\varpi) \in Multiterminal - Cut \iff \varpi \in Max - Cut \iff f(\varpi) \in Multiterminal - Cut$. А это по определению означает, что $Max - Cut \leqslant_p Multiterminal - Cut$. □

Теперь проверим оба условия из определения NP-полной задачи:

1. $Multiterminal - Cut \in \mathsf{NP}$, так как сертификатом в алгоритме верификации будет необходимый разрез C, чью стоимость можно проверить, пройдясь по нужным рёбрам графа, за полиномиальное от размера графа время.

2. Согласно лемме выше, $Max - Cut \leqslant_p Multiterminal - Cut$, а так как $Max - Cut \in \mathsf{NPC}$, то значит, $\forall\ \mathscr{A} \in \mathsf{NP} : \mathscr{A} \leqslant_p Max - Cut \leqslant_p Multiterminal - Cut$, что соответствует определению NP-трудного языка.

Оба условия выполняются, значит, $Multiterminal - Cut \in \mathsf{NPC}$. □

Сложность Задачи сегментации

Переформулируем условие Задачи сегментации:

Segmentation

Ввод: $G = (V,\, E)$ – неориентированный граф, L – метки сегментов, $|L| = l$, $P = (p_{uv})$ – матрица рёбер $(u,\, v) \in E$, множество штрафов $\{c_p(a)\}$, число k

Вывод: Ответ на вопрос: "Существует ли классификация – разбиение на множества, обозначенные различными метками – $\kappa : V \to L$, такая, что $\sum\limits_{p \in V} c_p\big(\kappa(p)\big) + \sum\limits_{\substack{u,\ v \in V \\ \kappa(u) \neq \kappa(v)}} p_{uv} \leqslant k$??"

Теперь оценим сложность данной задачи – докажем, что она является NP-полной.

Теорема 7. *Segmentation* $\in$ NPC

Доказательство. Для начала докажем следующую лемму:

Лемма. $Multiterminal - Cut \leqslant_p Segmentation$

Доказательство. Приведём функцию f, вычислимую за полиномиальное время, которая сводит задачу $Multiterminal - Cut$ к задаче о поиске наилучшей классификации вершин графа. Граф $G = (V, E)$ будет тот же. Матрицу $P = (p_{uv})$ заполняем так: $p_{uv} = w(u, v)$, $(u, v) \in E)$ – веса рёбер. Множество меток L мы получим, попросту обозначив за него множество терминальных вершин: $L = T$. Осталось получить $\{c_p(a)\}$. Это множество штрафов мы заполняем по следующему правилу:

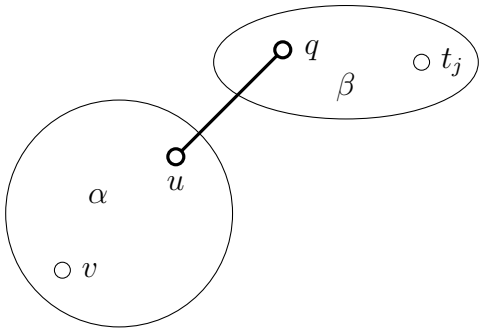
$$c_p(a) = \begin{cases} \sum\limits_{(u,\ v) \in E} w(u, v), \text{ если } p \in T \text{ и } a \neq \kappa(p) \\ 0, \text{ иначе} \end{cases}$$

То есть, в случае, когда две терминальные вершины в одном сегменте, мы делаем штраф, равный сумме весов всех рёбер графа – очень большая величина. В ином случае штрафа нет.

Зададим классификацию κ следующим образом:

Пусть $v \in V$ принадлежит компоненте связности, в которую входит вершина $t_i \in T$. Тогда определим $\kappa(v) = t_i$. Покажем, что для каждой вершины $G(C)$ будет определён свой сегмент при классификации.

Предположим, что при разбиении вершин на классы в компоненте связности вершины v – α – не оказалось ни одной вершины $t \in T$.



Рассмотрим два случая:

- Пускай у нас в компоненте α также есть какая-то вершина u, такая, что в процессе разбиения на сегменты мы удалили ребро (u, q), где q – вершина компоненты связности β. Пускай в β имеется вершина $t_j \in T$. Тогда, по построению, $\kappa(q) = t_j$. Но так как в компоненте α нет терминальных вершин, то значит, оставив ребро (u, q), мы не нарушим условия задачи – t_j не станет связной с какой-либо другой терминальной вершиной. В то же время, если не удалять ребро (u, q) (не добавлять в разрез), то разрез C будет меньше оного в случае удаления данного ребра. Следовательно, такой вершины $u \in \alpha$ существовать не могло – разрез был бы не минимальным – и случай невозможен.

- Пускай такой вершины u не существует, тогда α всегда была отдельной компонентой связности (в исходном же неразрезанном графе). Тогда доопределим нашу классификацию κ на данной компоненте: $v \in \alpha \Rightarrow \kappa(v) = t_\alpha$ – присвоили свою метку всем вершинам компоненты, получив новый сегмент. Так как α всегда была отдельной компонентой, то условие задачи не нарушится от доопределения κ. Более того, это никак не повлияет на размер разреза C.

То есть, если существует компонента α, в которой нет ни одной терминальной вершины $t \in T$, мы присваиваем данной компоненте α метку t_α, и $L = L \cup \{t_\alpha\}$.

На этом мы закончили сведение. Ясно, что f является вычислимой за полиномиальное время – единственное, что нужно сделать нового из поданных на вход функции данных, это множество $\{c_p(a)\}$ и доопределение меток L, что делается за полином от числа рёбер и вершин соответственно.

Покажем теперь, что сведение корректно, то есть, что размер минимального многотерминального разреза исходном графе G равен весу разбиения пикселей в полученном графе для сегментации. Иными словами, $\varpi \in Multiterminal - Cut \iff f(\varpi) \in Segmentation$:

- $\varpi \in Multiterminal - Cut \implies f(\varpi) \in Segmentation$

  Пусть для задачи ϖ существует разрез C – минимальный многотерминальный разрез в G. Покажем, что тогда обязательно существует классификация того же веса в Задаче сегментации.

  Заметим, что каждая вершина принадлежит своей компоненте связности. Значит, любые две терминальные вершины $t_i \neq t_j \in T$ не связны в $G(C)$. А это в свою очередь означает, что все $c_p(a) = 0$ и $\sum\limits_{p \in V} c_p\big(\kappa(p)\big) = 0$.

  Кроме того, по факту, $\sum\limits_{\substack{u,\ v \in V \\ \kappa(u) \neq \kappa(v)}} p_{uv}$ – это сумма весов рёбер, соединяющих вершины из разных компонет связности, то бишь их разных сегментов. Следовательно, размер разреза

  $$|C| = \sum_{\substack{u,\ v \in V \\ \kappa(u) \neq \kappa(v)}} p_{uv}$$

  – и это же число равно сумме весов рёбер, за которые мы платим штраф при сегментации. Значит, размер разреза совпадает с весом разбиения при сегментации, который также минимален, откуда $f(\varpi) \in Segmentation$.

- $f(\varpi) \in Segmentation \implies \varpi \in Multiterminal - Cut$

  Пускай для задачи $f(\varpi) \in Segmentation$ имеется классификация κ такая, что задаёт минимальный вес разбиения

  $$\sum_{p \in V} c_p\big(\kappa(p)\big) + \sum_{\substack{u,\ v \in V \\ \kappa(u) \neq \kappa(v)}} p_{uv}$$

  Поскольку имеется разбиение, то значит, все терминальные вершины лежат в попарно разных сегментах:

  $$\forall\ t_i \neq t_j \in T : \kappa(t_i) \neq \kappa(t_j) \implies \sum_{p \in V} c_p\big(\kappa(p)\big) = 0$$

  – никакие две вершины с разными метками не являются свзяными, и штраф за них платить не придётся.

  В таком случае, вершины распределятся по компонентам следующим образом: V_i – компонента связности графа G, такая, что

  $$V_i = \{v \mid \kappa(v) = t_i\} \implies C = \{(u,\, v) \in E \mid \kappa(u) \neq \kappa(v)\} \implies |C| = \sum_{\substack{u,\ v \in V \\ \kappa(u) \neq \kappa(v)}} p_{uv}$$

– то есть, размер разреза в точности равен сумме весов рёбер, соединяющих вершины из разных компонент связности, что в свою очередь равно суммарному штрафу за рёбра между пикселями разных сегментов. Следовательно, размер минимального многотерминального разреза в точности равен весу разбиения при классификации вершин, и $\varpi \in Multiterminal - Cut$.

Таким образом, мы привели функцию f, вычислимую за полиномиальное время, такую, что размер минимального многотерминального разреза исходном графе G равен весу разбиения пикселей в полученном графе для сегментации, то есть
$\varpi \in Multiterminal - Cut \iff f(\varpi) \in Segmentation$. А это по определению означает, что $Multiterminal - Cut \leqslant_p Segmentation$. □

Теперь проверим оба условия из определения NP-полной задачи:

1. $Segmentation \in \mathsf{NP}$, так как сертификатом в алгоритме верификации будет необходимая классификация κ, суммарный штраф которой мы можем посчитать и проверить за полиномиальное время.

2. Согласно лемме выше, $Multiterminal - Cut \leqslant_p Segmentation$, а так как $Multiterminal - Cut \in \mathsf{NPC}$, то значит, $\forall\ \mathscr{A} \in \mathsf{NP} : \mathscr{A} \leqslant_p Multiterminal - Cut \leqslant_p Segmentation$, что соответствует определению NP-трудного языка.

Оба условия выполняются, значит, $Segmentation \in \mathsf{NPC}$. □

Приближённое решение Задачи сегментации

Сформулируем теперь условие задачи сегментации в виде задачи поиска самого решения.

Segmentation

Ввод: $G = (V,\ E)$ – неориентированный граф, L – метки сегментов, $|L| = l$, $P = (p_{uv})$ – матрица рёбер $(u,\ v) \in E$, множество штрафов $\{c_p(a)\}$

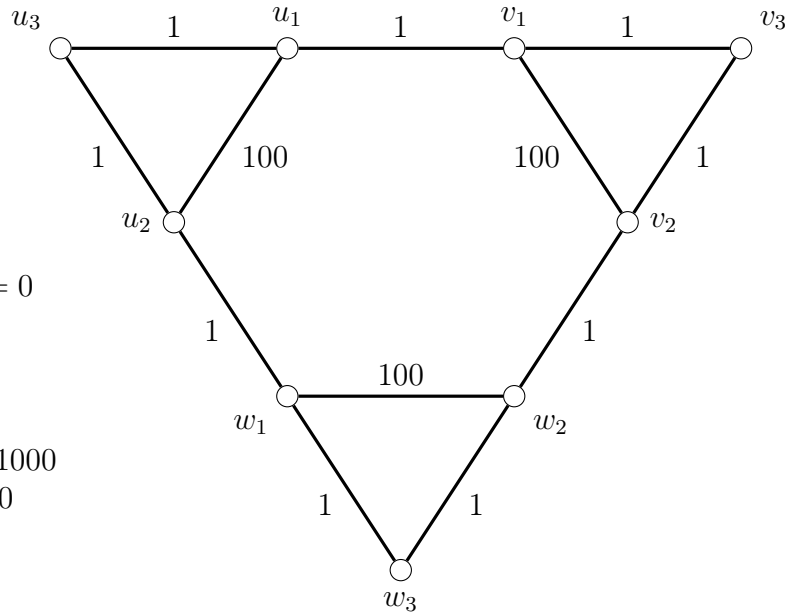
Вывод: Классификация – разбиение на множества, обозначенные различными метками – $\kappa : V \to L$, такая, что $\sum\limits_{p \in V} c_p\big(\kappa(p)\big) + \sum\limits_{\substack{u,\ v \in V \\ \kappa(u) \neq \kappa(v)}} p_{uv} \to \min$

Решим данную задачу для конкретного примера, который понадобится нам позднее. Пусть у нас есть такой граф, на котором определены межкпиксельные штрафы (веса рёбер) p_{st}, $(s, t) \in E$, и штрафы за принадлежность компоненте $c_p(a)$, $p \in V$, $a \in L = \{\textbf{red}, \textbf{green}, \textbf{blue}\}$:

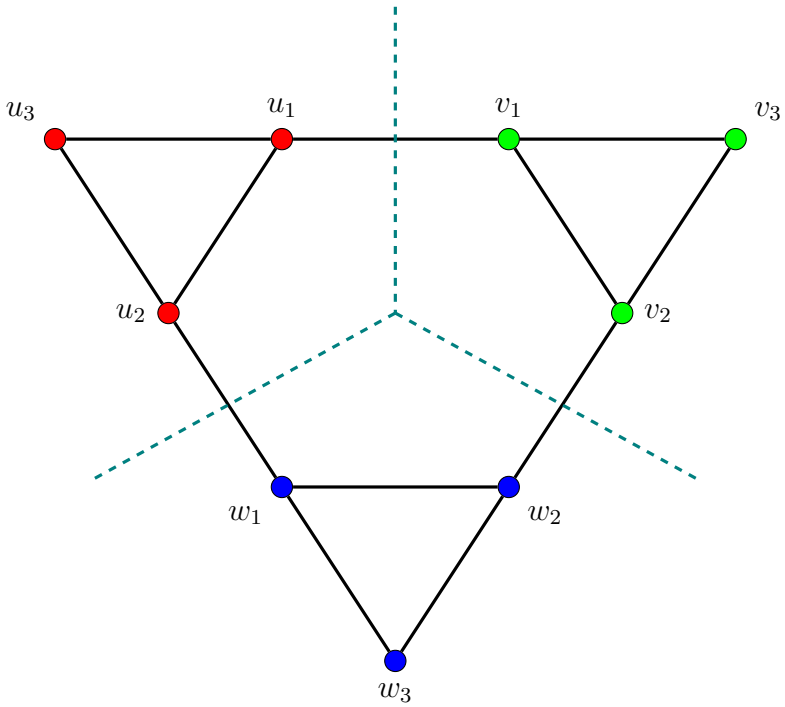
$c_{u_i}(\textbf{red}) = 0$
$c_{v_i}(\textbf{green}) = 0$
$c_{w_i}(\textbf{blue}) = 0$

$c_{w_1}(\textbf{red}) = c_{v_1}(\textbf{red}) = 0$
$c_{u_1}(\textbf{green}) = c_{w_2}(\textbf{green}) = 0$
$c_{v_2}(\textbf{blue}) = c_{u_2}(\textbf{blue}) = 0$

$c_{w_2}(\textbf{red}) = c_{v_2}(\textbf{red}) = 1000$
$c_{u_2}(\textbf{green}) = c_{w_1}(\textbf{green}) = 1000$
$c_{v_1}(\textbf{blue}) = c_{u_1}(\textbf{blue}) = 1000$

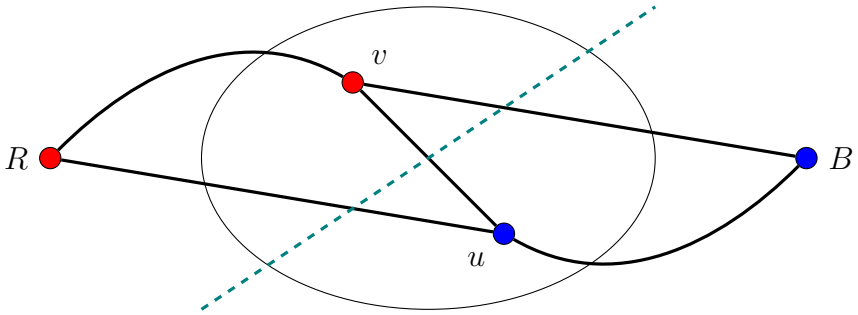


В данном случае оптимальным решением будет разбиение: $u_i \in \textbf{red}$, $v_i \in \textbf{green}$, $w_i \in \textbf{blue}$. Разрез пересекут всего 3 ребра веса 1, и стоимость сегментации будет равна 3:



Рассмотрим частный случай этой задачи. Пусть у нас имеется две метки: $L = \{\textbf{red}, \textbf{blue}\}$. Тогда ясно, что по факту перед нами стоит задача о разбиении вершин графа на два множества так, чтобы стоимость разбиения была минимальной. А это ни что иное, как **Задача о минимальном разрезе** – $Min-Cut$, к которой сводится частный случай задачи сегментации.

Сводим задачу мы так: сам граф $G = (V, E)$ со штрафами p_{st} оставляем. Нужно, однако, учитывать ещё компонентные штрафы $c_p(a)$. Чтобы их учесть, делаем 2 фиктивные вершины R и B, и соединяем все остальные вершины с данными двумя. Вершина R всегда будет в **red**-компоненте, вершина B - в **blue**-компоненте. Пусть у нас имеется вершина $v \in V$. Тогда делаем вес ребра (R, v) равным $c_v(\textbf{blue})$, а вес ребра (B, v) равным $c_v(\textbf{red})$. Тогда размер минимального разреза действительно будет совпадать с весом разбиения – рёбра p_{st} также будут пересекать разрез (или не пересекать). При этом, если $v \in \textbf{red}$, то разрез будет пересекать ребро (B, v) веса $c_v(\textbf{red})$ – платим тот самый штраф. Аналогично, если $v \in \textbf{blue}$, то разрез будет пересекать ребро (R, v) веса $c_v(\textbf{blue})$ – платим штраф за принадлежность компоненте.



Таким образом мы получим минимальный разрез размера, равного весу разбиения в аналогичной задаче сегментации. Как нам известно, $Min - Cut$ можно решить за полиномиальное от размера графа время. То есть, для двух меток задачу можно эффективно решить. Попробуем теперь свести задачу сегментации с любым количеством меток к поиску решения $Min - Cut$, чтобы получить пусть и приближённое, но полиномиальное решение.

Первая попытка

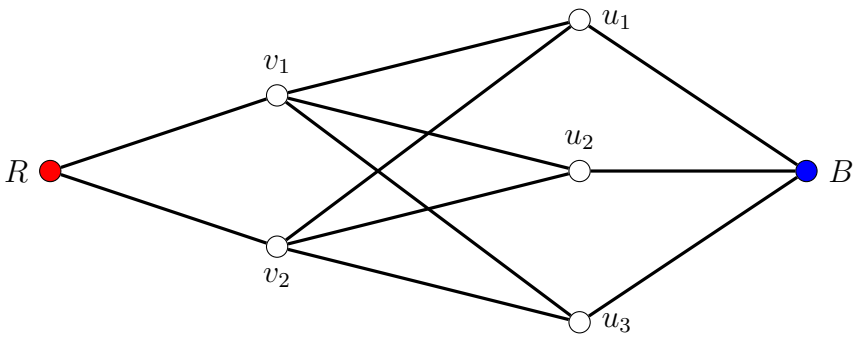
Вернёмся к общему случаю и будем пытаться решить задачу приближённо методом **локального поиска**. Идея заключается опять же в переходах между соседями путём перебрасывания вершин между компонентами.

Для начала определим отношение соседства. Пускай у нас есть некое решение – классификация κ. Тогда соседними решениями будем считать множество

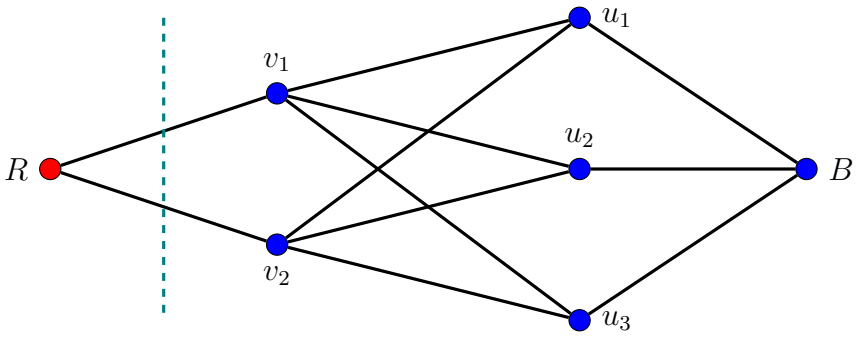
$$N(\kappa) = \{\kappa' \mid \exists\, v \in V : \forall\, u \in V,\, u \neq v \Rightarrow \kappa(u) = \kappa'(u)\}$$

– мы фиксируем вершину v, переклассифицируем все вершины так, что свою метку сменит только одна вершина v, а все остальные вершины останутся в своих классах. Иначе говоря, мы перекидываем одну вершину в другую компоненту, все остальные оставляя без изменений.

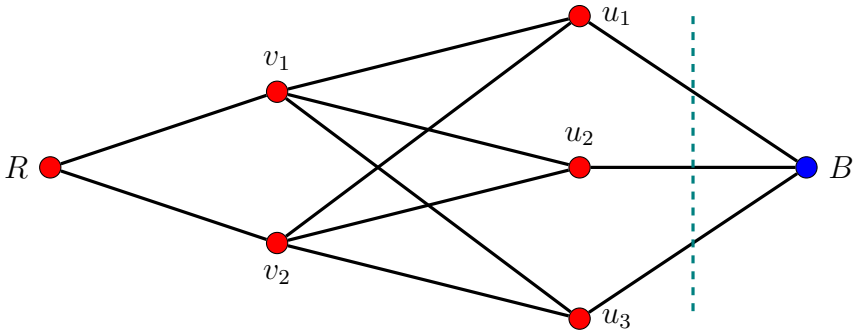
Увы, но такое отношение соседства не оптимально. Рассмотрим граф:



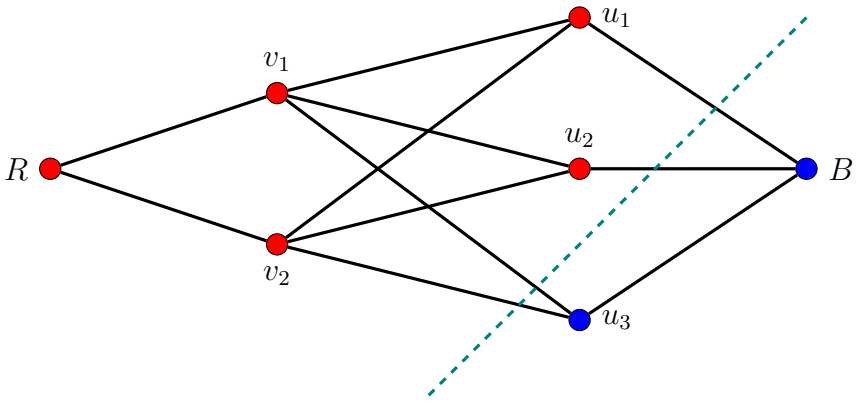
Пускай вообще все рёбра имеют вес 1, как межпиксельные, так и компонентные. В таком случае оптимальным решением будет поместить вершину R в **red**-компоненту, а все остальные вершины – в **blue**-компоненту – разрез пересечёт всего два ребра и будет иметь размер 2:



Однако, предположим, что в процессе поиска оптимального решения методом локального поиска – переходя к соседям-разрезам разбиениям меньшей стоимости – мы оказались в таком разрезе:



Это локальный минимум – какую бы вершину мы не перекинули из **red**-компоненты в **blue**-компоненту, мы увеличим размер разреза на 1 – вместо одного ребра, пересекающего разрез, у нас появится два:



Таким образом, локальный поиск не сможет выйти из данного решения, а оно не оптимально. Значит, такое отношение соседства не сможет привести нас к наилучшей классификации.

Вторая попытка

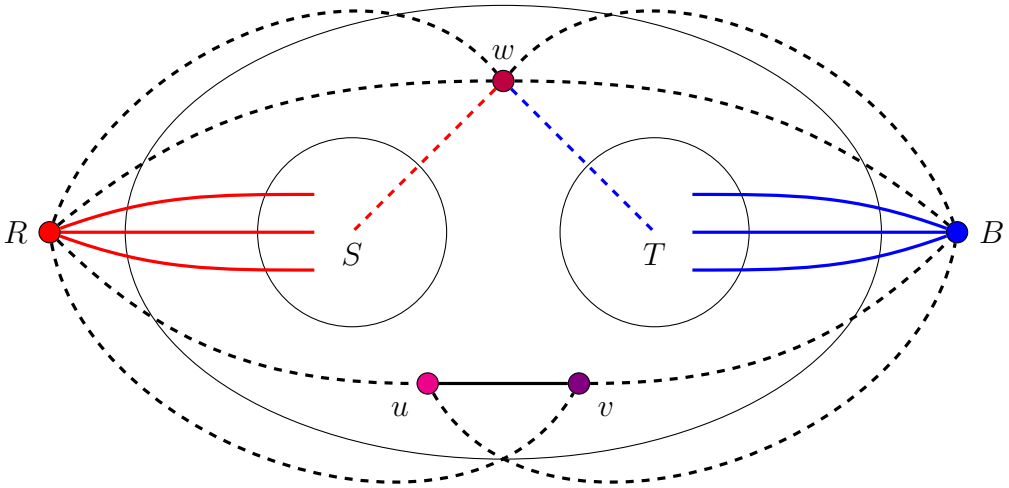
Изменим отношение соседства. Пускай у нас есть некое решение – классификация κ. Тогда соседними решениями будем считать множество

$$N(\kappa) = \{\kappa' \mid \exists\ a, b \in L,\ a \neq b : \forall\ d \in L,\ a \neq d \neq b \Rightarrow \forall\ v \in V : \kappa(v) = d \iff \kappa'(v) = d\}$$

– иначе говоря, мы можем менять метки только у тех вершин, что имеют метки a или b, причём менять их можем опять же только на a или b, в то время как метки всех остальных вершин остаются без изменений. Меняются таким образом только компоненты, отвечающие меткам a и b.

Заметим, что при $|L| = 2$ для любой классификации соседними будут в точности все возможные разбиения вершин на 2 множества, коих $O(2^{|V|})$ штук. И в процессе поиска решения при $|L| > 2$ можем оказаться в ситуации, что практически все вершины разбиты по 2-м множествам – случай с $|L| = 2$. Однако, возможно, что задачу можно решить за полиномиальное время, имея при этом экспоненциальное число соседей.

Собственно, да, так и есть. Пускай у нас имеется граф G, и мы рассматриваем компоненты S и T, отвечающие меткам s и t соответственно. Создаём вершины R – **red**-компонента, и B – **blue** компонента. Соединяем с соответствующими вершинами в компонентах S, T и вне их.



Нужно найти оптимального соседа. Сделаем круче – сразу же найдём наилучшего из них, то есть такого, при котором суммарный штраф за разбиение при возможности изменения только меток s и t будет наименьшим: будем на данном графе теперь искать минимальный разрез, но только для вершин из компонет S и T. Стоит отметить, что для остальных вершин (тех, у которых метки не s и не t) ничего не меняется, вне зависимости от того, куда перешли смежные им вершины между компонентами S и T:

- Вершина w не принадлежит компонентам S и T. Пускай она соединена ребром с какой-то вершиной из S. Тогда если данная вершина перешла в компоненту T, то межпиксельный штраф за принадлежность разным компонентам – p_{st} – для данного ребра не изменится и всё равно будет считаться, ведь w как была в другой компоненте, так и осталась.

- Если какие-то две вершины u и v были соединены ребром между собой, то от изменения компонент S и T штраф за ребро (u, v) так и будет считаться.

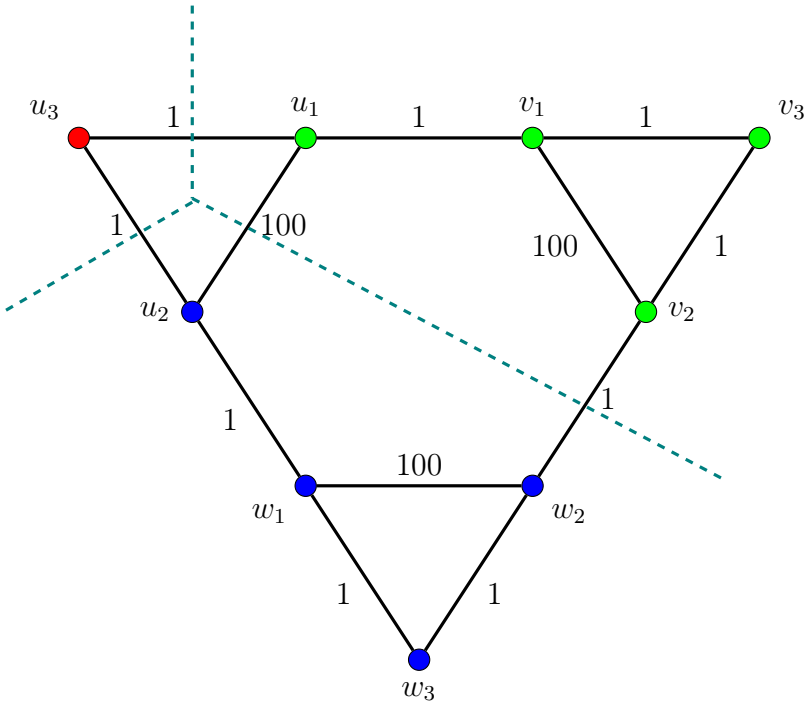
Если $Min - Cut$ выдал лучшее решение, чем текущее разбиение – тогда заменяем наше разбиение на полученное. Если получилось худшая или такая же классификация – то оставляем разбиение для s и t без изменений, считая таковое наилучшим.

$Min - Cut$ находит оптимальное разбиение для пары меток s и t за некоторое время K. Ясно, что для того, чтобы найти оптимальное решение, нам потребуется перебрать все пары меток, которых:

$$\binom{|L|}{2} = \frac{|L|!}{2! \cdot (|L| - 2)!} = \frac{|L| \cdot (|L| - 1)}{2} = O(|L|^2)$$

Значит, одна итерация алгоритма локального поиска займёт время K, всего запускаем $O(|L|^2)$ раз, и итоговое время работы будет $O(K \cdot |L|^2)$ – полиномиальное время от размера входа (задачи сегментации). Алгоритм и вправду полиномиален!

Как бы то ни было прискорбно, но и данный алгоритм не может нам гарантировать оптимальное решение, так как тоже может упасть в локальный минимум. Рассмотрим граф, который разбирали в самом начале лекции, и предположим, что наш алгоритм попал в ситуацию, когда разбиение вышло следующим:



Вес данного разбиения (размер разреза) равен 103 (штрафы за вершины u_1 и u_2, что сменили метки, не изменяются – для новых меток они также равны 0), а как мы помним, оптимальное решение имеет стоимость 3. Значит, текущая классификация не является оптимальной. Однако, мы никуда из данной ситуации выйти не сможем – это локальный минимум. Попытавшись изменить расположения вершин в какой-то паре компонент из {**red**, **green**, **blue**} мы только увеличим размер разреза или не изменим его – ребро веса 100 так и так будет пересекать разрез (проверьте данные факты самостоятельно, перебрав случаи).

Таким образом, локальный поиск не сможет выйти из данного решения, а оно не оптимально. Значит, и такое отношение соседства не сможет привести нас к наилучшей классификации.

Третья попытка

Ещё раз изменим отношение соседства. Пускай у нас есть некое решение – классификация κ. Тогда соседними решениями будем считать множество

$$N(\kappa) = \left\{ \kappa' \mid \exists\, a \in L : \forall\, v \in V \Rightarrow \begin{bmatrix} \kappa'(v) = a \\ \kappa'(v) = \kappa(v) \end{bmatrix} \right\}$$

– мы фиксируем метку a, переклассифицируем все вершины так, что любая вершина графа $v \in V$ может либо поменять свою метку на a, либо оставить без изменений. Иначе говоря, мы можем менять метки вообще для всех вершин, но только путём замены текущей метки на a. То есть, к a мы можем только «докинуть» вершин, а из всех остальных можем только «выкидывать».

Ответим на два интересующих нас вопроса – как в таком случае эффективно находить соседей, и насколько же хорошо получившееся решение.

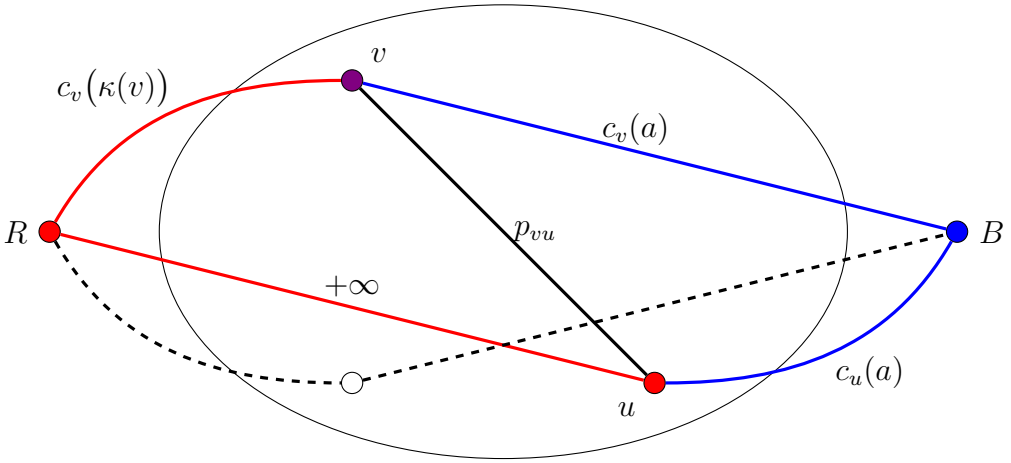
Сразу стоит отметить, что соседей $O(|L| \cdot 2^{|V|})$, при чём при любом числе меток – для каждой вершины можем определить, лежит ли она в компоненте a или нет, так для каждой метки. В то же время, покажем, что сей факт не особо нам навредит – мы будем снова сводить поиск лучшего соседа к задаче о минимальном разрезе, тем самым получая полиномиальное решение. Опишем, как мы сможем это сделать.

Имеем текущую классификацию κ. Пускай у нас имеется граф $G = (V, E)$ со штрафами p_{st} – сохраним данные рёбра (s, t) в графе. Нужно, однако, учитывать ещё компонентные штрафы $c_p(a)$. Мы рассматриваем компоненты **red** и **blue**. **red**-компонента отвечает всем вершинам, имеющим метку a, **blue**-компонента же отвечает всем вершинам с отличной от a меткой – $\neg a$. Создаём вершины R **red**-компоненты, и B **blue** компоненты, и соединяем каждую вершину $v \in V$ рёбрами с данными вершинами. Покажем, как мы работаем со штрафами за принадлежность компоненте.

Пусть у нас имеется вершина $v \in V$. Она соединена ребром с B, отвечающей $\neg a$ компоненте. Тогда делаем вес ребра $(B,\ v)$ равным $c_v(a)$ – если у v метка a ($v \in$ **red**), то разрез будет пересекать ребро (B, v) веса $c_v(a)$ – платим тот самый штраф.

v также соединена ребром с R, отвечающей a компоненте. Тогда делаем вес ребра $(R,\ v)$ равным $c_v\big(\kappa(v)\big)$. Действительно, если у v метка $\neg a$ ($v \in$ **blue**), то мы должны заплатить штраф за то, что у v как раз таки та другая метка. Но эта метка равна значению функции классификации от вершины v – $\kappa(v)$. Тогда разрез будет пересекать ребро (R, v) веса $c_v\big(\kappa(v)\big)$ – платим тот самый штраф.

Стоит отметить, что наше отношение соседства запрещает вершинам из a менять свою компоненту. Пусть у нас имеется какая-то вершина $u \in V$ такая, что $\kappa(u) = a$. Она соединена ребром с R, отвечающей a компоненте, и с B, отвечающей $\neg a$ компоненте. $u \in$ **red** постоянно, значит, ребро $(R,\ u)$ не может пересекать разрез – компоненты a и R всегда совпадают и равны **red**. Следовательно, чтобы гарантировать, что ребро $(R,\ u)$ не пересечёт разрез, сделаем вес данного ребра равным $+\infty$. Вес ребра $(B,\ u)$, соответственно, будет равен $c_u(a)$, что много меньше $+\infty$, поэтому такое ребро будет в разрезе.

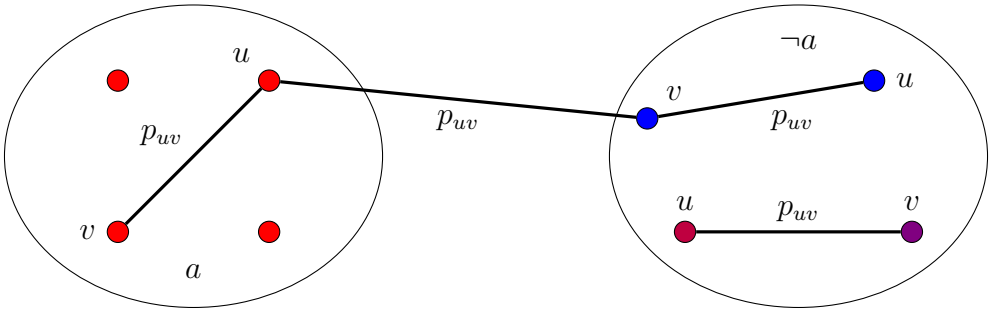


Чтобы убедиться, что на таком графе поиск минимального разреза для компонент **red** и **blue** даст оптимальное решение, рассмотрим теперь подробнее, как учитывается вес самих рёбер p_{st} в постоенном графе.

У нас есть две компоненты – a и $\neg a$.

Пускай у нас имеются вершины u и v. Они соответствующим образом соединены между собой рёбрами. При этом вершины u и v могут иметь как одну и ту же метку, так и разные. Такое разбиение мы получили для классификации κ.

Пускай мы перешли к новой классификации κ' в процессе поиска наилучшего решения. При этом вершины u и v могли сменить свои компоненты на a.



Посмотрим, какие случаи из возможных мы учли:

хорошо $\kappa(u) = a = \kappa(v)$

Если две вершины лежат в компоненте a, то они не сменят компоненту при переходе к соседней классификации. Поэтому мы как не платили штраф p_{uv}, так и не будем.

хорошо $\kappa(u) = a \neq \kappa(v)$

В этом случае в исходном разбиении κ мы будем платить штраф за ребро (u, v). В новом разбиении у нас может возникнуть 2 варианта:

1. Обе вершины u и v оставят метку без изменений – тогда как платили штраф p_{uv}, так и будем, ведь (u, v) осталось в $(a, \neg a)$ разрезе.
2. v сменит метку на a. Тогда мы не будем платить штраф за ребро (u, v), так как $u \in \mathbf{red} \ni v$.

хорошо $a \neq \kappa(u) = \kappa(v)$

В этом случае в исходном разбиении κ мы не будем платить штраф за ребро (u, v). В новом разбиении у нас может возникнуть 2 варианта:

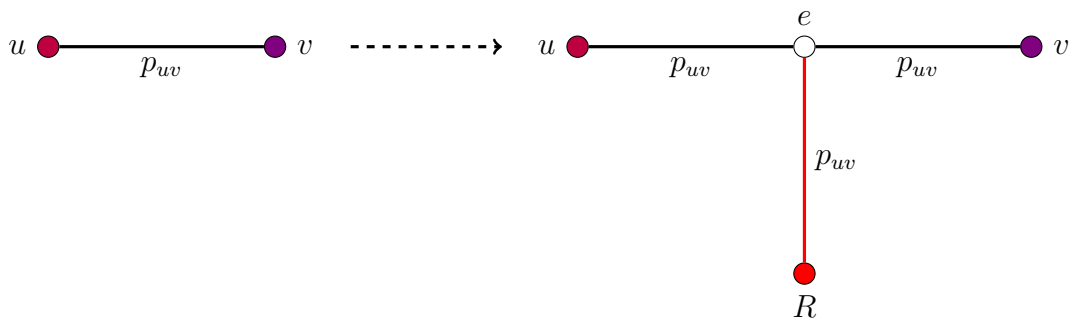
1. Если обе вершины u и v сменят метку на a, то нам не придётся платить штраф за ребро (u, v), так как оно соединяет вершины одной компоненты.
2. Если вершина u сменит метку на a, то ребро (u, v) пересечёт разрез, и мы будем платить платить штраф p_{uv}, ибо u и v оказались в разных компонентах.

плохо $a \neq \kappa(u) \neq \kappa(v) \neq a$

В этом случае в исходном разбиении κ мы должны платить штраф за ребро (u, v). Однако, оно не пересекает разрез – ведь ни одна из вершин не лежит в a. Кроме того, в ново разбиении возникают варианты:

1. Если вершина u сменит метку на a, то ребро (u, v) пересечёт разрез, и мы будем платить платить штраф p_{uv}, ибо u и v оказались в разных компонентах – это мы учли.
2. Обе вершины u и v оставят метку без изменений – тогда как платили не штраф p_{uv}, так и будем, ведь (u, v) осталось в $(a, \neg a)$ разрезе. Но это некорректно, так как данные вершины имеют разные метки. То есть, мы не учитываем такой расклад событий.

Чтобы учитывать «плохой» случай, будем при переходе между решениями $\kappa \to \kappa'$ производить преобразование рёбер графа (u, v), таких, что $a \neq \kappa(u) \neq \kappa(v) \neq a$. Пускай у нас имеется ребро (u, v) данного типа. Превратим его в следующий объект, который будем называть **гаджет-ребром**:



То есть, «вставляем» между вершинами u и v новую вершину e, соединяя данную вершину с u и v рёбрами того же веса, что и ребро (u, v). Также соединяем e с вершиной **red**-компоненты R, вес такой же – p_{uv}.

Положим, что мы перешли от решения κ к решению κ'. Метка вершины e будет зависеть от того, как мы переклассифицируем вершины в новом разбиении, так как алгоритм $Min - Cut$ определит её в ту компоненту, в которой штраф будет наименьшим.

Покажем, что теперь штрафы «плохого» случая будут учтены корректно.

- $\kappa'(u) = a = \kappa'(v)$

  Теперь $u \in$ **red** $\ni v$, и мы не должны платить штрафа p_{uv}. Определим, куда попадёт e. Если $e \in$ **red**, то ни одно ребро гаджет-ребра не будет пересекать разрез, и штраф составит 0. Если же $e \in$ **blue**, то все 3 ребра будут в разрезе, и штраф составит $3 \cdot p_{uv}$. Следовательно, $Min - Cut$ определит e в **red** компоненту (в силу меньшего штрафа), и $\kappa'(e) = a$. Штрафа при этом мы не заплатим.

- $\kappa'(u) \neq a = \kappa'(v)$

  Одна вершина имеет метку a, другая имеет метку $\neg a$. Тогда мы обязаны платить штраф только один раз, за то, что ребро (u, v) пересекает разрез. Определим, куда попадёт e. Если $e \in$ **red**, то только одно ребро гаджет-ребра – (u, e) – пересечёт разрез, и штраф составит p_{uv}. Если же $e \in$ **blue**, то разрез будут пересекать (e, v) и (R, e), и штраф составит $2 \cdot p_{uv}$. Следовательно, $Min - Cut$ определит e в **red** компоненту и в этом случае, и $\kappa'(e) = a$. При этом мы заплатим штраф за ребро (u, e), равный штрафу за (u, v), как и должны.

- $a \neq \kappa'(u) \neq \kappa'(v) \neq a$

  Вершины остались в своих компонентах, причём в различных. Тогда мы обязаны платить штраф только один раз, за то, что ребро (u, v) пересекает разрез. Определим, куда попадёт e. Если $e \in$ **red**, то разрез будут пересекать (e, v) и (u, e), и штраф составит $2 \cdot p_{uv}$. Если же $e \in$ **blue**, то только одно ребро гаджет-ребра – (R, e) – пересечёт разрез, и штраф составит p_{uv}. Следовательно, $Min - Cut$ определит e в **blue** компоненту и в этом случае, и $\kappa'(e) = \neg a$. Полагаем, что $Min - Cut$ присудит e метку одной из вершин u или v – не важно, какой именно, так как разрез (**red**, **blue**) рёбра (e, v) и (u, e) не пересекут. Штраф придётся платить только за ребро (R, e), равный штрафу за (u, v), как и требуется.

В итоге, при помощи построения гаджет-рёбер мы сможем учитывать в весе разбиения штрафы даже за те рёбра, что не пересекают разрез (**red**, **blue**) по умолчанию при переходе между решениями $\kappa \to \kappa'$. Тогда итоговый размер разреза, полученный $Min - Cut$, после данного преобразования будет в точности равен весу разбиения κ'. Следовательно, вот такое отношение соседства уже позволяет нам получить действительно оптимальное решение.

Оценим время работы нашего поиска решения задачи. Построение гаджет-рёбер для каждого ребра, а также создание нужных рёбер для $R \in$ **red** и $B \in$ **blue** занимает на каждой

итерации алгоритма $O(|E|)$ времени. $Min-Cut$ находит оптимальное разбиение для пары меток a и $\neg a$, для которых строится разрез (**red**, **blue**), за некоторое время K, полиномиальное от размера графа. А всего итераций алгоритма поиска оптимальной классификации мы совершаем $|L|$ – по числу возможных вариантов выбора $a \in L$. Итоговое время займёт $O\big(|L|\cdot K\cdot|E|\big)$, а оно полиномиально зависит от размера входа задачи сегментации. Следовательно, наш алгоритм полностью удовлетворяет нашим требованиям и действительно находит решение задачи сегментации за полиномиальное время.

Оценка качества приближения

Проверим, насколько хорошее приближение нам даёт наш алгоритм. Обозначим как $C(\kappa)$ суммарную стоимость нашей классификации. Наш алгоритм работает так:

Алгоритм 12 Segmentation

Ввод: $G = (V, E)$ – неориентированный граф, L – метки сегментов, $|L| = l$, $P = (p_{uv})$ – матрица рёбер $(u, v) \in E$, множество штрафов $\{c_p(a)\}$

function CHEAPER_NEIGHBOUR(κ)
  for $a \in L$ **do**
    $N =$ **create_(red, blue)_graph**(G, κ, a) – строим описанное выше сведение
    $(U, W) = Min - Cut$(**red**, **blue**) – находим минимальный разрез
    if $|(U, W)| < C(\kappa)$ **then**
      $\kappa' = \kappa$ – создаём новую классификацию κ', по умолчанию она совпадает с κ
      for $u \in U$ **do**
        $\kappa'(u) = a$ – заменяем метки у вершин из **red**-компоненты (отвечающей a)
      $\kappa = \kappa'$ – обновляем разбиение для продолжения работы
  return κ

выбрать случайное разбиение κ
while $\exists\ \kappa' \mid \kappa' =$ **cheaper_neighbour**(κ) **do**
  $\kappa = \kappa'$
return получившееся разбиение κ

Оценим качество работы алгоритма. Пусть κ^* – это оптимальное разбиение, а κ – какая-то классификация, которую вернул алгоритм **Segmentation**.

Зафиксируем некую метку $a \in L$. Обозначим как $V^*(a)$ множество всех вершин с меткой a в классификации κ^*:

$$V^*(a) = \{v \in V \mid \kappa^*(v) = a\}$$

Определим функцию κ_a – это функция-классификации, соседняя с κ, работающая так:

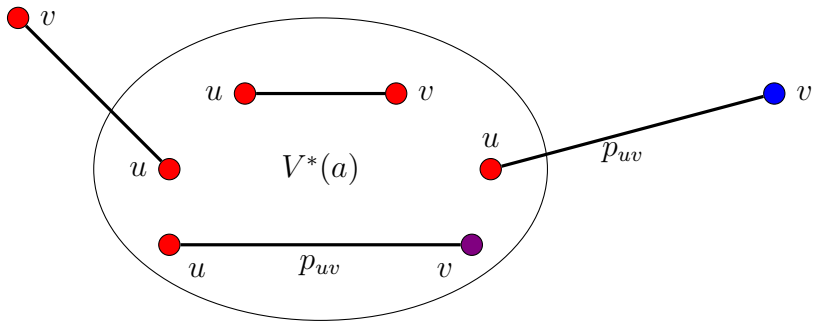
$$\kappa_a(v) = \begin{bmatrix} a, \text{ если } v \in V^*(a) \\ \kappa(v) \text{ иначе} \end{bmatrix}$$

Ясно, что κ дешевле, чем κ_a – ведь наш алгоритм вернул именно разбиение κ, как обладающее наименьшим среди соседей весом. Тогда $C(\kappa_a) \geq C(\kappa) \iff C(\kappa_a) - C(\kappa) \geq 0$.

Заметим, что κ_a и κ отличаются теми и только теми вершинами, что сменили свою метку с $\kappa(v)$ на a в $\kappa_a(v)$. Тогда можно ограничить разность выше сверху:

$$C(\kappa_a) - C(\kappa) \leqslant \sum_{v \in V^*(a)} c_v(a) - c_v\big(\kappa(v)\big) + \text{разность межпиксельных штрафов}$$

Поймём, как можно ограничить разность суммарных штрафов за рёбра между вершинами разных меток. Для этого посмотрим, какие штрафы у нас были в κ, и какие получились в κ_a. Возьмём ребра $(u,\, v) \in E$ и посмотрим, какие случаи будут учтены в каждом решении:



$V \setminus V^*(a)$ Суммарный штраф за рёбра, соединяющие вершины различных меток, на вершинах u и v таких, что и $u \notin V^*(a)$, и $v \notin V^*(a)$, у κ и κ_a совпадает. Тогда в разности штрафов он сократится.

κ_a В данном случае возникают новые рёбра, связанные с вершинами из $V^*(a)$. Ясно, что при классификации κ_a все вершины $u \in V^*(a)$ будут лежать в компоненте a. Однако, метку a могут иметь некоторые вершины не из $V^*(a)$. Оценим штраф за них грубо, утверждая, что при данной классификации все вершины $v \notin V^*(a)$ будут иметь метку, отличную от a. Тогда суммарный штраф за рёбра между вершинами разных меток, таких, что одна вершина лежит в $V^*(a)$, составит не более чем $\sum\limits_{\substack{(u,\, v)\in E \\ u\in V^*(a) \\ v\notin V^*(a)}} p_{uv}$

κ В данном случае мешают рёбра, связанные с вершинами из $V^*(a)$. Рассмотрим их поподробнее. Пусть вершина $u \in V^*(a)$, и при этом $\kappa(u) = a$. Тогда в исходной классификации в разрез будут входить все рёбра $(u,\, v)$, где $\kappa(v) \neq \kappa(u)$. Заметим, что все штрафы за такие рёбра мы уже ссуммировали при рассмотрении κ_a. Значит, чтобы получить разницу штрафов, из всей суммарной разности штрафов мы дополнительно должны вычесть величину $\sum\limits_{\substack{(u,\, v)\in E \\ u\in V^*(a) \\ \kappa(u)\neq\kappa(v)}} p_{uv}$.

Получили следующую оценку:

$$0 \leqslant C(\kappa_a) - C(\kappa) \leqslant \sum_{v\in V^*(a)} c_v(a) - c_v\big(\kappa(v)\big) + \sum_{\substack{(u,\, v)\in E \\ u\in V^*(a) \\ v\notin V^*(a)}} p_{uv} - \sum_{\substack{(u,\, v)\in E \\ u\in V^*(a) \\ \kappa(u)\neq\kappa(v)}} p_{uv}$$

Нас будет интересовать правая часть неравенства.

Такую стоимость классификации мы получили для одной метки a. Просуммируем по всем меткам $a \in L$:

$$0 \leqslant \sum_{a\in L} \left(\sum_{v\in V^*(a)} c_v(a) - c_v\big(\kappa(v)\big) + \sum_{\substack{(u,\, v)\in E \\ u\in V^*(a) \\ v\notin V^*(a)}} p_{uv} - \sum_{\substack{(u,\, v)\in E \\ u\in V^*(a) \\ \kappa(u)\neq\kappa(v)}} p_{uv} \right)$$

Перенесём отрицательные слагаемые влево:

$$\underbrace{\sum_{a\in L}\sum_{v\in V^*(a)} c_v\big(\kappa(v)\big)}_{\text{сумма по всем вершинам}} + \underbrace{\sum_{a\in L}\sum_{\substack{(u,\ v)\in E\\ u\in V^*(a)\\ \kappa(u)\neq\kappa(v)}} p_{uv}}_{\text{перебираем все рёбра для разных меток}} \leqslant \underbrace{\sum_{a\in L}\sum_{v\in V^*(a)} c_v(a)}_{\text{сумма только по меткам из } V^*(a)} + \underbrace{\sum_{a\in L}\sum_{\substack{(u,\ v)\in E\\ u\in V^*(a)\\ v\notin V^*(a)}} p_{uv}}_{\text{сумма рёбер по всем меткам}}$$

Оценим правую часть сверху. Заметим, что при подсчёте $\sum\limits_{a\in L}\sum\limits_{\substack{(u,\ v)\in E\\ u\in V^*(a)\\ v\notin V^*(a)}} p_{uv}$ мы учитываем каждое ребро не более чем дважды: пускай у нас есть ребро $(u,\ v)$, где $u \in V^*(a)$, и $v \in V^*(b)$. Тогда штраф p_{uv} мы посчитаем, сначала рассмотрев ребро $(u,\ v)$ для $a \in L$, $u \in V^*(a)$, и впоследствии рассмотрев его же для $b \in L$, $v \in V^*(b)$ – два раза, ибо в оптимальной раскраске данные вершины точно будут иметь различные метки. Тогда это число не более чем в 2 раза больше, чем межпиксельный штраф для оптимальной классификации κ^*. Однако, при этом для суммы $\sum\limits_{a\in L}\sum\limits_{v\in V^*(a)} c_v(a)$ мы посчитаем в точности величину, не большую (вообще говоря равную) компонентного штрафа для κ^* – мы суммируем штрафы только за вершины, принадлежащие компонентам оптимального разбиения. Тогда всю сумму мы можем смело оценить сверху удвоенной стоимостью отимальной классификации:

$$\sum_{a\in L}\sum_{v\in V^*(a)} c_v(a) + \sum_{a\in L}\sum_{\substack{(u,\ v)\in E\\ u\in V^*(a)\\ v\notin V^*(a)}} p_{uv} \leqslant 2\cdot C(\kappa^*)$$

В то же время, мы можем оценить левую часть снизу стоимостью разбиения κ, что вернул наш алгоритм: сумма $\sum\limits_{a\in L}\sum\limits_{v\in V^*(a)} c_v\big(\kappa(v)\big)$ в точности равна покомпонентному штрафу для нашего разбиения κ. Однако, при подсчёте $\sum\limits_{a\in L}\sum\limits_{\substack{(u,\ v)\in E\\ u\in V^*(a)\\ \kappa(u)\neq\kappa(v)}} p_{uv}$ мы учитываем каждое два раза: пускай у нас есть ребро $(u,\ v)$, где $u \in V^*(a)$, и $v \in V^*(b)$. Тогда штраф p_{uv} мы посчитаем, сначала рассмотрев ребро $(u,\ v)$ для $a \in L$, $u \in V^*(a)$, и впоследствии рассмотрев его же для $b \in L$, $v \in V^*(b)$ – два раза. Следовательно, данная величина точно не меньше стоимости получившейся классификации:

$$C(\kappa) \leqslant \sum_{a\in L}\sum_{v\in V^*(a)} c_v\big(\kappa(v)\big) + \sum_{a\in L}\sum_{\substack{(u,\ v)\in E\\ u\in V^*(a)\\ \kappa(u)\neq\kappa(v)}} p_{uv}$$

В итоге мы можем оценить эффективность нашего алгоритма:

$$C(\kappa) \leqslant 2\cdot C(\kappa^*)$$

– отличается не более чем в 2 раза. То есть, мы получили 2 приближение.

Оценим теперь время работы нашего алгоритма, показав, что оно псевдополиномиально. Выполнение функции **cheaper_neighbour** занимает полиномиальное время – мы выполняем на каждом шаге функцию **create_(red, blue)_graph**, которая работает за полином – (из предыдущего раздела), а также перебираем $O(|V|)$ вершин, переклассифицируя одну метку. А шагов мы совершаем $|L|$. Тогда время работы **cheaper_neighbour** равно $O(|L|\cdot|L|\cdot K\cdot|E|\cdot|V|) = O\big(|L|^2\cdot K\cdot E\cdot V\big)$.

Сколько нам нужно итераций, прежде чем мы получим результат? Стоит отметить, что если каждый раз уменьшать штраф $C(\kappa)$ для каждого последующего κ на 1, то мы можем получить в худшем случае экспоненциальное время работы.

Попробуем немного ослабить точность алгоритма, чтобы получить полиномиальный алгоритм. Пусть имеется некая константа $\varepsilon > 0$, и будем брать такие разрезы $|(U, W)|$, что меньше $C(\kappa)$ не более, чем в $1 - \dfrac{\varepsilon}{3|L|}$ раз. В таком случае, повторяя рассуждения выше для нового параметра (подобно тому, как мы делали для оценки $Max - Cut$), качество нашего решения не слишком сильно изменится:

$$C(\kappa) \leqslant (2 + \varepsilon) \cdot C(\kappa^*)$$

Проверим время работы. Пускай наш алгоритм совершил $\dfrac{|L|}{\varepsilon}$ итераций. Тогда за это время разрез уменьшится в $\left(1 - \dfrac{\varepsilon}{3|L|}\right)^{\frac{|L|}{\varepsilon}}$ раз, что можно оценить сверху:

$$\left(1 - \frac{\varepsilon}{3|L|}\right)^{\frac{|L|}{\varepsilon}} = \left(\left(1 - \frac{\varepsilon}{3|L|}\right)^{\frac{3|L|}{\varepsilon}}\right)^{\frac{1}{3}} \leqslant \{\text{так как } 1 - x \leqslant e^{-x}\} \leqslant \left(\frac{1}{e}\right)^{\frac{1}{3}} = \alpha \Rightarrow \text{константа}$$

То есть, через $\dfrac{|L|}{\varepsilon}$ итераций решение улучшится в константное число раз. А так как на каждом таком шаге длины $\dfrac{|L|}{\varepsilon}$ мы уменьшаем размер разреза на сумму штрафов, которым отвечают некоторые рёбра графа, то значит, всего шагов может быть:

$$\text{шагов} \leqslant \log_\alpha \sum_{(u,\ v) \in E} p_{uv}$$

Тогда общее число итераций нашего алгоритма составит

$$\frac{|L|}{\varepsilon} \cdot \log_\alpha \sum_{(u,\ v) \in E} p_{uv}, \text{ где } \alpha = \left(\frac{1}{e}\right)^{\frac{1}{3}}$$

– полиномиальное от размера входа. Но тогда и сам наш алгоритм будет занимать лишь полиномиальное время работы, так как все остальные функции также работают за полином.

Таким образом, мы получили полиномиальный от размера входа алгоритм поиска оптимальной классификации для задачи сегментации, дающий 2-приближённое решение.
А это довольно-таки неплохо.

Задача кластеризации

Постановка задачи

Пускай у нас имеется множество данных. Мы хотим разбить данные на множества, элементов, имеющих схожие значения по некоторым характеристикам. Иными словами, по массиву входных данных мы хотим получить разбиение оных на множества похожим элементов. В этом заключается **Задача кластеризации**.

Clustering

Ввод: Множество объектов $P = \{p_1, \ldots, p_n\}$, число $k \in \mathbb{N}$, $k \leqslant n$, функция расстояния $d : P \times P \to \mathbb{R}$, $d \geqslant 0$, обладающая следующими свойствами:

1. $\forall\ p \in P : d(p, p) = 0$
2. $\forall\ p_i, p_j \in P : d(p_i, p_j) = d(p_j, p_i)$ – симметричность

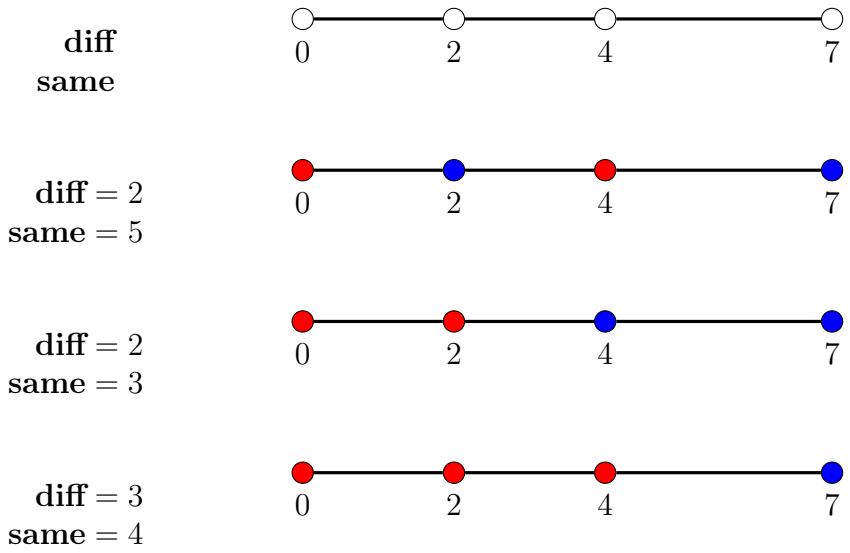
Вывод: Разбиение P на k множеств $C_1, \ldots, C_k$ – кластеров, где

- $C_1 \cup \ldots \cup C_k = \bigcup\limits_{i=1}^{k} C_i = P$
- $\forall\ i \neq j : C_i \cap C_j = \varnothing$

такое, что оно удовлетворяет некоторому уловию. Примеры таких условий:

1. Минимальное расстояние между элементами разных множеств должно быть максимально: $\min\limits_{i \neq j} d(p', p'') \mid p' \in C_i,\ p'' \in C_j \longrightarrow \max$ – обозначим это число как **diff**
2. Максимальное расстояние между элементами из одного множества должно быть минимально: $\max\limits_{i} d(p', p'') \mid p' \in C_i,\ p'' \in C_i \longrightarrow \min$ – обозначим это число как **same**

Решим задачу для следующего примера: пускай у нас есть 4 числа на числовой прямой: $P = \{0, 2, 4, 7\}$. Функция d будет возвращать попросту расстояние между числами на прямой. Необходимо разбить числа на два множества – **red** и **blue**.



Как видим, для **diff** и **same** оптимальными являются разные разбиения: **diff** максимально в последнем варианте кластеризации, в то время как **same** минимально во втором. Первый же вариант проигрывает для каждого условия. То есть, для различных условий разбиения оптимальное решение может различаться, и не факт, что решение окажется наилучшим по всем условиям.

Рассмотрим подробнее, как можно решить задачу для условия максимизации **diff**.

Решение diff при помощи Алгоритма Крускала

Пускай у нас имеется множество объектов, и нам необходимо найти оптимальную **diff**-кластеризацию. Изначально у нас все объекты в различных множествах – n штук. Идея решения заключается в следующем: будем брать два самых близких – $d(p_i, p_j) \to \min$ – объекта, и объединять их в одно множество. Затем ищем следующую пару самых близких объектов, и тоже их объединяем в одно множество. Если один из объектов уже лежит в каком-то множестве (кластере), то объединяем данные множества в одно. Таким образом, мы будем продолжать объединять объекты и множества до тех пор, пока не достигнем необходимого количества множеств k и при этом не сможем объеденить какие-то два объекта под одно множество, не уменьшив при этом общее число всех кластеров.

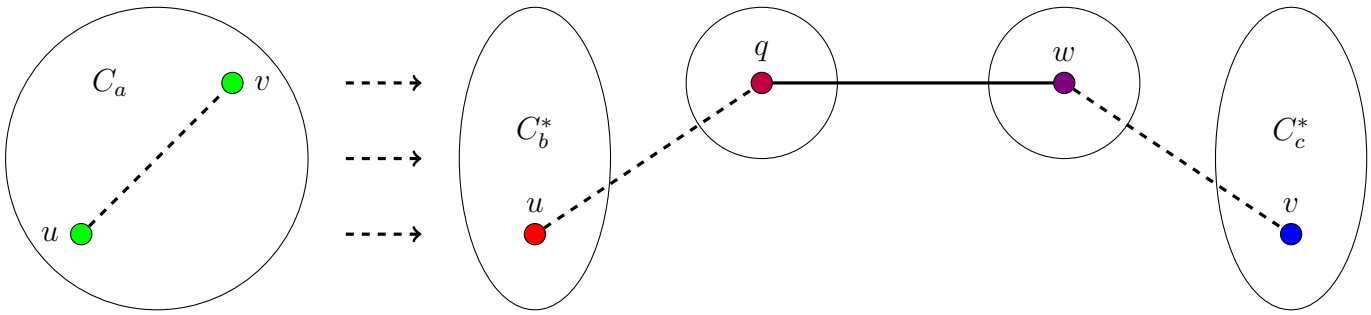
Если говорить уже о конкретной реализации решения, то в данной задаче мы будем пользоваться **Алгоритмом Крускала**. Действительно, сведём задачу кластеризации к задаче поиска минимального остовного дерева в графе: представим наши элементы $\{p_1, \ldots, p_n\}$ как вершины некоего графа, а функция $d(p_i, p_j)$ являет собой функцию весов рёбер графа. Будем при этом пользоваться структурой данных **система непересекающихся множеств**, чтобы оптимизировать время работы. В таком случае минимальное остовное дерево в получившемся графе будет в точности совпадать с оптимальной кластеризацией в исходной задаче.

Стоит отметить, что асимптотика алгоритма Крускала в графе на n вершинах и m рёбрах равна $O(m \cdot \log m)$. Однако, при сведении граф получится полным, и $m = n^2$. Тогда время работы составит $O(n^2 \cdot \log n)$.

Теорема 8. *Алгоритм Крускала корректно решает задачу кластеризации.*

Доказательство. Пусть алгоритм Крускала вернул разбиение $C_1, \ldots, C_k$. Предположим, что наше разбиение не оптимально, и существует более хорошее оптимальное разбиение $C_1^*, \ldots, C_k^*$.

Оптимальное разбиение отлично от нашего, значит, существует пара вершин u, v, которые в нашем разбиении лежат в одном кластере, а в оптимальном – в разных кластерах (иначе решения бы совпадали). Пускай $u \in C_a$ и $v \in C_a$ в исходном разбиении, а в оптимальном разбиении $u \in C_b^*$, $v \in C_c^*$.



Поскольку в нашем решении вершины u и v были в одном кластере, то значит, они соединены каким-то путём. Рассмотрим путь u—v. На этом пути существует такое ребро (q, w), что вершины q и w находятся в разных кластерах в оптимальном решении – если бы такого ребра не существовало, то значит, на пути из u в v для любого ребра данного пути (q', w') вершины q' и w' находятся в одном кластере. Но тогда все вершины данного пути должны лежать в одном кластере, в то время как сами вершины u и v в оптимальном решении принадлежат разным кластерам – противоречие. Значит, такое ребро найдётся. Отметим, что в нашем решении q и w также в C_a.

Взглянем подробнее на данное ребро (q, w). Заметим, что межкластерное расстояние **diff*** в оптимальном решении есть максимальная длина ребра, соединяющего ближайшие вершины различных кластеров. В таком случае, оно точно не больше длины ребра (q, w):

$$\textbf{diff*} \leqslant d(q, w)$$

Возьмём теперь наше разбиение и ребро (s, t) – $k-1$-е самое тяжёлое ребро в минимальном остовном дереве. То есть, если бы нужно было разбиение на 1 кластер больше, то алгоритм Крускала добавил бы именно это ребро. Будем рассматривать шаг работы алгоритма Крускала, на котором произошло добавление ребра (q, w), тем самым объединяя вершины q и w. Стоит отметить, что на тот момент вершины q и w лежали в разных кластерах. Кроме того, все рёбра, что остались недобавленными в итоговом разбиении, не были добавлены и на данный момент. Так как алгоритм Крускала добавляет к минимальному остову ребро наименьшего веса, то значит, для ребра (s, t), оставленного недобавленным на данном шаге, выполнено, что

$$d(q, w) \leqslant d(s, t)$$

Заметим, что межкластерное расстояние **diff** в нашем решении есть максимальная длина ребра, соединяющего ближайшие вершины различных кластеров в итоговом получившемся по алгоритму Крускала разбиении. Но тогда величина **diff** равна длине ребра (s, t), откуда в силу вышеприведённого неравенства

$$d(q, w) \leqslant \textbf{diff}$$

Следовательно,

$$\textbf{diff*} \leqslant d(q, w) \leqslant \textbf{diff}$$
$$\Downarrow$$
$$\textbf{diff*} \leqslant \textbf{diff}$$

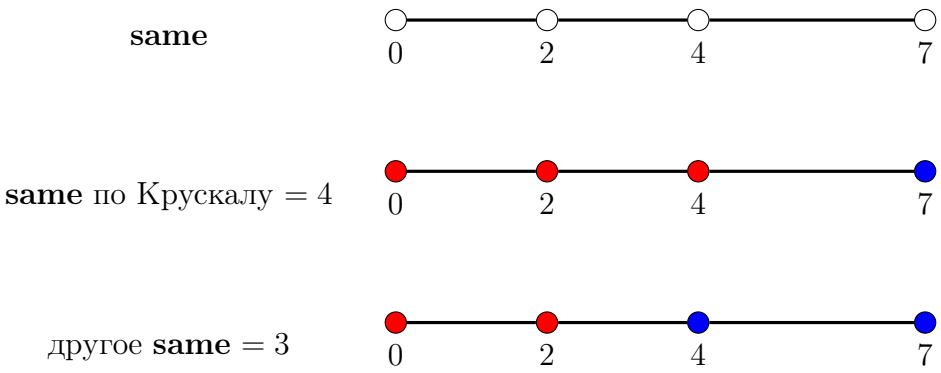
Для решения задачи кластеризации необходимо найти наилучшее **diff**, то есть максимальное. Но мы получили, что межкластерное расстояние в решении алгоритма Крускала оказалось не больше, чем в некотором более хорошем оптимальном решении задачи. Пусть оно строго меньше – $\textbf{diff*} < \textbf{diff}$ – тогда данное оптимальное решение не является таковым, ибо хуже решения алгоритма Крускала. Противоречие.

Таким образом, разбиение, получаемое алгоритмом Крускала, действительно является наилучшим, и алгоритм корректно решает задачу кластеризации.

□

Задача кластеризации с same условием

Стоит отметить, что алгоритм Крускала уже не решает данную задачу правильно. Покажем это на примере:



Иначе говоря, алгоритмом Крускала данную задачу не решить. Более того, вообще говоря задача кластеризации с **same** условием является NP-полной. Докажем данный факт. Для начала перепишем условие задачи кластеризации в виде верификационной задачи (то есть задачи, проверяющей на принадлежность языку):

Clustering

Ввод: Множество объектов $P = \{p_1, \ldots, p_n\}$, число $k \in \mathbb{N}$, $k \leqslant n$, некоторое число $D \in \mathbb{R}$ и функция расстояния $d : P \times P \to \mathbb{R}$, $d \geqslant 0$, обладающая следующими свойствами:

1. $\forall\ p \in P : d(p, p) = 0$
2. $\forall\ p_i, p_j \in P : d(p_i, p_j) = d(p_j, p_i)$

Вывод: Ответ на вопрос: "Существует ли разбиение P на k множеств $C_1, \ldots, C_k$ – кластеров, где

- $C_1 \cup \ldots \cup C_k = \bigcup\limits_{i=1}^{k} C_i = P$
- $\forall\ i \neq j : C_i \cap C_j = \varnothing$

такое, что для любого кластера C_i и $\forall\ p_l \in C_i$, $\forall\ p_m \in C_i$ выполнено, что $d(p_l, p_m) \leqslant D$??"

В данном случае D представляет из себя как раз то самое **same**.

Будем доказывать NP-полноту задачи кластеризации путём сведения к ней известной NP-полной **Задачи о раскраске графа**.

Coloring

Ввод: Граф $G = (V, E)$, число $k \in \mathbb{N}$

Вывод: Ответ на вопрос: "Можно ли раскрасить данный граф G в k цветов ??"

Теорема 9. ***same*** $-$ *Clustering* $\in$ NPC.

Доказательство. Для начала докажем следующую лемму:

Лемма. *Coloring* $\leqslant_p$ ***same*** $-$ *Clustering*

Доказательство. Приведём функцию f, вычислимую за полиномиальное время, которая сводит задачу *Coloring* к задаче о поиске оптимального **same**-разбиения, строя нужный набор объектов P и задавая необходимые условия для кластеризации.

Имеем граф $G = (V, E)$. Тогда положим $P = V$ – множеством объектов будет множество вершин графа G. Число k необходимых кластеров равно числу цветов, в которые мы хотим покрасить граф. Работу функции расстояния d на объектах (вершинах) $u, v \in P\ (\in V)$ зададим следующим образом:

$$d(u, v) = \begin{cases} 0, \text{ если } u = v \\ 1, \text{ если } (u, v) \notin E \\ 2, \text{ если } (u, v) \in E \end{cases}$$

Число D делаем равным 1. На этом сведение закончено.

Покажем, что сведение корректно, то есть, граф G является k-раскрашиваемым если и только если существует разбиение множества P на кластеры, удовлетворяющее всем условиям задачи **same** $-$ *Clustering*.

Пусть существует раскраска в k цветов в исходном графе. По определению k-раскрашиваемого графа, между вершинами одного цвета нет рёбер. Заметим, что множество вершин одного цвета являет собой кластер для задачи **same** $-$ *Clustering*. Действительно, поскольку между вершинами одного цвета нет рёбер, то значит для любого кластера C_i и $\forall\ p_l \in C_i$, $\forall\ p_m \in C_i$ выполнено, что $d(p_l, p_m) = 1 \leqslant D = 1$ по построению функции d. Но тогда каждому из k цветов соответствует удовлетворяющий условиям **same** $-$ *Clustering* кластер. При этом между вершинами разного цвета будут рёбра, и соответствующие вершины будут лежать на расстоянии $2 > D$ друг от друга, что не нарушает ограничения на d – ведь вершины разного цвета, а значит, и кластеры различны. Следовательно, у нас существует подходящее разбиение множества $P = V$ на k кластеров.

Пусть теперь существует необходимое разбиение множества P на k непересекающихся кластеров, таких, что для любого кластера C_i и $\forall\ p_l \in C_i$, $\forall\ p_m \in C_i$ выполнено, что $d(p_l, p_m) \leqslant$

$D = 1$. Так как для любых двух вершин u и v из кластера C_i выполнено, что $d(u, v) \leqslant 1$, то по построению $d(u, v) = 0$, откуда $u = v$, или же $d(u, v) = 1$. В последнем случае мы получаем, что $(u, v) \notin E$ – ребро (u, v) отсутствует в исходном графе. Следовательно, между вершинами u и v графа G, такими, что в разбиении на k кластеров они обе попали в один кластер, ребра нет. Тогда, покрасив данные вершины в один цвет, мы не нарушим условия k-раскрашиваемости графа, ведь между рёбрами одинакового цвета не будет ребра. В таком случае, покрасив вершины каждого из k кластеров в соответствующий цвет, мы получим раскраску графа G в k цветов. Кроме того, все вершины w и q, находящиеся на расстоянии $d(w, q) = 2$ будут соединены ребром. Но так как $d(u, v) \leqslant D = 1$ для любых u и v из одного кластера, то значит, данные w и q будут лежать в разных кластерах и соответственно будут иметь различные цвета. Но тогда ребро (w, q) соединяет вершины разных цветов. Значит наша раскраска в k цветов является корректной.

Таким образом, мы привели функцию f, вычислимую за полиномиальное время и строящую по задаче $Coloring$ задачу $\mathbf{same}-Clustering$, такую, что граф G является k-раскрашиваемым если и только если существует разбиение множества P на кластеры, удовлетворяющее всем условиям задачи $\mathbf{same} - Clustering$. А это по определению означает,
что $Coloring \leqslant_p \mathbf{same} - Clustering$. □

Теперь проверим оба условия из определения NP-полной задачи:

1. $\mathbf{same} - Clustering \in \mathsf{NP}$, так как сертификатом в алгоритме верификации будет сама кластеризация вершин графа, а проверка выполнения **same**-условия занимает $O(E)$ времени – полиномиальное от размера входа время.

2. Согласно лемме выше, $Coloring \leqslant_p \mathbf{same} - Clustering$, а так как $Coloring \in \mathsf{NPC}$, то значит, $\forall\, \mathscr{A} \in \mathsf{NP} : \mathscr{A} \leqslant_p Coloring \leqslant_p \mathbf{same}-Clustering$, что соответствует определению NP-трудного языка.

Оба условия выполняются, значит, $\mathbf{same} - Clustering \in \mathsf{NPC}$. □

Приближённое решение при помощи метода центров

Будем искать приближённое решение задачи кластеризации с **same** условием. Запишем условие задачи кластеризации по нахождению оптимального **same** расстояния, предварительно задав метрику неравенства треугольника. Введём для каждого кластера C_i величину

$$D_{C_i} = \max_{u,\ v \in C_i} d(u, v)$$

– назовём её **диаметром** кластера C_i. Также для каждой точки p и кластера C введём

$$dist(p, C) = \min_{c \in C} d(p, c)$$

– расстояние от точки p до кластера C.

Clustering

Ввод: Множество объектов $P = \{p_1, \ \ldots, p_n\}$, число $k \in \mathbb{N}$, $k \leqslant n$, функция расстояния $d : P \times P \to \mathbb{R}$, $d \geqslant 0$, обладающая следующими свойствами:

1. $\forall\ p \in P : d(p, p) = 0$
2. $\forall\ p_i, p_j \in P : d(p_i, p_j) = d(p_j, p_i)$ – симметричность
3. $\forall\ p_i, p_j, p_k \in P : d(p_i, p_j) \leqslant d(p_i, p_k) + d(p_k, p_j)$ – неравенство треугольника

Вывод: Разбиение P на k непересекающихся множеств $C_1, \ \ldots, \ C_k$ – кластеров, такое, что максимальный среди всех диаметров кластеров C_i был минимален: $D_{C_i} = \max\limits_{u,\ v \in C_i} d(u, v) \longrightarrow \min$ – этим числом и будем считать **same**

Стоит отметить, что сведение $Coloring \leqslant_p \mathbf{same} - Clustering$ работает корректно и при введённой метрике неравенства треугольника.

Построим алгоритм, приближённо решающий задачу кластеризации. Идея такова, что в процессе работы алгоритма мы будем делать некоторые объекты **центрами** – специальными объектами, каждый центр связан со своим кластером.

Алгоритм 13 Clustering

Ввод: Множество объектов $P = \{p_1, \ldots, p_n\}$, число $k \in \mathbb{N}$, $k \leqslant n$, функция расстояния d

1: выбираем случайный объект $c_1 \in P$ – это будет первый центр
2: $C = \{c_1\}$ – множество центров
3: **for** $i = 2$ **to** k **do**
4: $\quad c_i = \underset{p \in P}{\mathbf{argmax}}\, dist(p, C)$ – выбираем самую удалённую от всех текущих центров
5: $\quad C = C \cup \{c_i\}$ – делаем данную точку центром
6: **for** $i = 1$ **to** k **do** $C_i = \left\{ p \mid \forall\, j \neq i : \left[\begin{array}{l} d(p, c_i) < d(p, c_j) \\ d(p, c_i) = d(p, c_j),\ i < j \end{array} \right. \right\}$ – разбиваем объекты на кластеры, добавляя объект в кластер самого близкого центра. Если расстояния от объекта до двух центров совпадают, то договоримся добавлять в тот, чей индекс меньше.
7: **return** получившееся разбиение $C_1, \ldots, C_k$

Полиномиальность данного алгоритма очевидна. Докажем, что он корректен.

Теорема 10. *Данный алгоритм вычисляет 2-приближённое решение. То есть,*

$$\forall\, [C_1, \ldots, C_k] : \max_{C_i^* \in [C_1^*, \ldots, C_k^*]} D_{C_i^*} \leqslant 2 \cdot \max_{C_i \in [C_1, \ldots, C_k]} D_{C_i}$$

где $[C_1^*, \ldots, C_k^*]$ – ***same****-кластеризация, что вернул алгоритм.*

Доказательство. Необходимо показать, что для любого другого разбиения C $\mathbf{same}*$ – диаметр C^*, будет не более чем 2 раза больше **same** – максимального диаметра разбиения C.

Будем рассматривать возвращённое алгоритмом разбиение C^*. Пусть $p = \underset{u \in P}{\mathbf{argmax}}\, dist(u, C^*)$ – самый удалённый от всех центров объект. Можно отметить, что он же является первым из недобавленных объектов-центров, то есть, если бы требовалось разбить P на $k+1$ кластер, то $c_{k+1}^* = p$. После разбиения p будет лежать в некотором кластере C_i^*. Обозначим за r расстояние между p и центром кластера, которому p принадлежит, то есть $r = d(p, c_i^*)$.

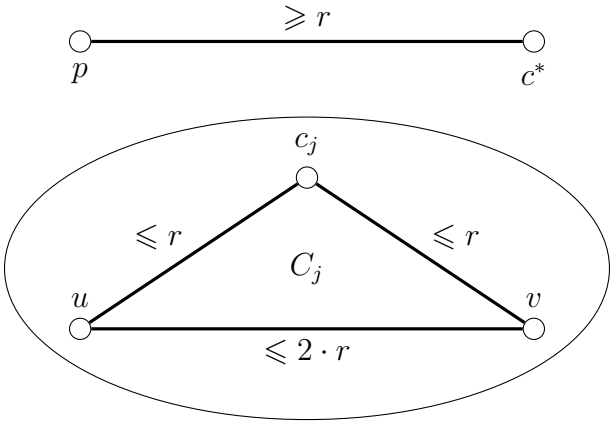
Оценим расстояние между центрами кластеров. Оно будет не меньше данного числа r:

$$\forall\, l, \forall\, m > l : d(c_m^*, c_l^*) \geqslant r$$

так как на тех шагах работы алгоритма, когда мы добавляли максимально удалённые от всех центров объекты, мы добавили именно c_m^*, а не p. Следовательно, такие центры c_m^* удалены от центров, добавленных на более ранних шагах алгоритма, на всяко не меньшее расстояние, чем $d(p, c_i^*) = r$.

Обозначим как $C_p^* = \{c_1^*, c_2^*, \ldots, c_k^*, p\}$ – множество всех оптимальных центров с объектом p. Из утверждения выше следует, что все объекты из C_p^* удалены друг от друга на расстояние не меньше r. Ясно, что для любого разбиения множества P хотя бы 2 объекта из C_p^* попадут в один и тот же кластер (так как $|C_p^*| = k+1 > k$ – по принципу Дирихле). Значит, при любой кластеризации **same** будет точно не меньше r.

Теперь будем рассматривать произвольное разбиение. Возьмём кластер C_j с центром c_j и объекты $u \in C_j$, $v \in C_j$. Объект p при этом в каком-то другом кластере с центром c^* (но вообще говоря кластер p не имеет значения, важно то, что он в одном кластере с $c^* \in C_p^*$).



Поскольку p – самый удалённый от центров объект, то значит, любой другой объект будет удалён от центра своего кластера уж точно не на большее расстояние. Но тогда по правилу треугольника расстояние между u и v будет не больше удвоенного максимального расстояния:

$$\forall\ p_i,\, p_j,\, p_k \in P : d(p_i,\, p_j) \leqslant d(p_i,\, p_k) + d(p_k,\, p_j)$$

$$\Downarrow$$

$$d(p,\, c) \geqslant r \Longrightarrow \begin{cases} d(u,\, c_j) \leqslant r \\ d(v,\, c_j) = d(c_j,\, v) \leqslant r \end{cases} \Longrightarrow d(u,\, v) \leqslant d(u,\, c_j) + d(c_j,\, v) \leqslant 2 \cdot r$$

Получается, что для любых двух объектов u, v из одного кластера расстояние $d(u,\, v)$ между ними будет не больше удвоенного максимального расстояния между кластером и объектом, что и даёт нам 2-приближённое решение. □

Возникает логичный вопрос: а существует ли ещё более хорошее приближённое решение? Ответим на него при помощи сведения $Coloring \leqslant_p \mathbf{same} - Clustering$, что мы проделали ранее. Расстояние задавалось так:

$$d(u,\, v) = \begin{cases} 0, \text{ если } u = v \\ 1, \text{ если } (u,\, v) \notin E \\ 2, \text{ если } (u,\, v) \in E \end{cases}$$

Предположим, что существует приближённый алгоритм решения задачи $\mathbf{same}-Clustering$, работающий за полином, такой, что даёт $2 - \varepsilon$-приближение, $\varepsilon > 0$. Тогда:

- Если существует k-раскраска графа $G = (V,\, E)$, то значит, существует кластеризация с максимальным диаметром 1. Но $2 - \varepsilon$-приближённый алгоритм даёт кластеризацию с диаметром не более $2 - \varepsilon$, а так как всего диаметр может принимать значения 0, 1 или 2, то значит, алгоритм всегда будет находить кластеризацию с диаметром не больше 1. Тогда мы тем самым найдём k-раскраску графа G, при чём за полиномиальное время.

- Если k-раскраски графа G не существует, то значит, кластеризации с диаметром менее 2 нет. Тогда алгоритм найдёт кластеризацию с диаметром 2.

Следовательно, если мы хотим найти k-раскраску графа G, то достаточно найти $2 - \varepsilon$-приближённое решение задачи кластеризации, что делается за полиномиальное время. Причём k-раскраска существует тогда и только тогда, когда существует решение с диаметром 1. Однако, мы знаем, что $Coloring \in \mathsf{NPC}$. Тем самым мы получили полиномиальное решение NP-трудной задачи. Тогда $\mathsf{P} = \mathsf{NP}$.

Последний факт пока что является открытой научной проблемой. Следовательно, неизвестно, существует ли приближённый алгоритм решения задачи $\mathbf{same} - Clustering$, работающий за полином, такой, что даёт $2 - \varepsilon$-приближение, $\varepsilon > 0$ – ведь мы не знаем, равны ли P и NP.

Задача коммивояжера

Рассмотрим известную NP-полную задачу – **Задачу коммивояжера**.

Travelling-Salesman

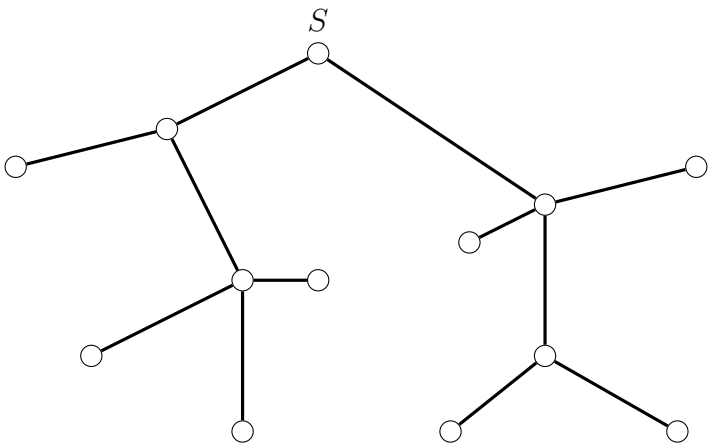
Ввод: Граф $G = (V, E)$ полный неориентированный граф ($|E| = O(|V|^2)$), функция весов $w : E \to \mathbb{R} \geqslant 0$, такая, что при этом выполнено неравенство треугольника:

$$\forall\ u, v, q \in V : w(u, v) \leqslant w(u, q) + w(q, v)$$

Вывод: Гамильтонов цикл минимального веса.

Обозначим как C^* искомый гамильтонов цикл. Найдём приближённое решение задачи.

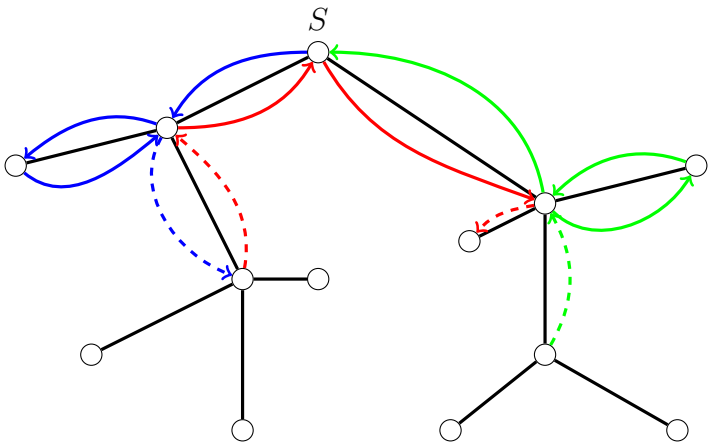
Для начала построим в графе G минимальное остовное дерево T (воспользовавшись тем же алгоритмом Крускала).



Ясно, что при этом суммарный вес оставшихся рёбер будет не больше, чем вес оптимального гамильтонова цикла, то есть

$$w(T) \leqslant w(C^*)$$

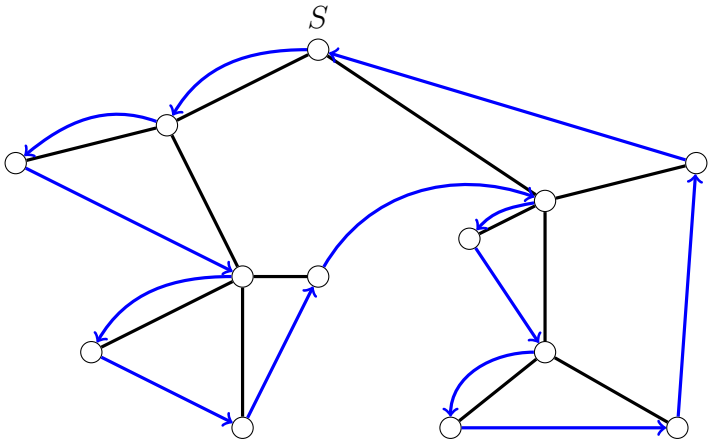
Будем обходить дерево по рёбрам, начиная с вершины S, методом поиска в глубину, то есть, спускаться по дереву пока есть такая возможность, а затем возвращаться обратно тем же путём, пока не появится новая развилка (непосещённая соседняя вершина):



Ясно, что по каждому ребру мы пройдём дважды – спускаясь ниже по дереву и возвращаясь обратно к вершине. По факту, это даёт нам очевидное неравенство:

$$2 \cdot w(T) \leqslant 2 \cdot w(C^*)$$

Сделаем простой путь из данного обхода дерева: будем вместо возвращения в вершину, откуда пришли, в случае невозможности продолжения дальнейшего спуска сразу же идти в следующую непосещённую вершину в порядке обхода DFS. Так как изначально граф был полным, то такой путь существует.



Получили какой-то гамильтонов цикл C – обход графа, являющийся простым циклом на основе минимального остовного дерева. Заметим, что в силу неравенства треугольника:

$$\forall\ u,\ v,\ q \in V : w(u,\ v) \leqslant w(u,\ q) + w(q,\ v)$$

полученный путь будет меньшего веса, нежели путь DFS, так как вместо возвращения в вершину мы сразу же идём к другому соседу той самой вершины, куда бы вернулись в случае обхода DFS, тем самым идя по ребру меньшего веса. Но тогда получаем, что

$$w(C) \leqslant 2 \cdot w\big(T\big) \leqslant 2 \cdot w\big(C^*\big)$$
$$\Downarrow$$
$$w(C) \leqslant 2 \cdot w\big(C^*\big)$$

То есть, данный алгоритм даёт нам 2-приближённое решение задачи коммивояжера.

Рандомизированные алгоритмы. Проверка равенства многочленов. Выполнимость булевых формул.

Пускай у нас есть два многочлена $A(x)$ и $B(x)$. Нам нужно установить, равны ли они, то есть, совпадает ли $A(x)$ с $B(x)$ с точностью до перестановки одночленов. Это утверждение эквивалентно тому, что $A(x)$ и $B(x)$ имеют полностью совпадающие множества решений. Сформулируем **Задачу равенства многочленов**.

Poly-Equal

Ввод: многочлены $A(x)$, $B(x)$ степеней d

Вывод: Ответ на вопрос: "$A(x) \equiv B(x)$??"

Решать данную задачу мы будем при помощи рандомизированных алгоритмов.

Алгоритм Монте-Карло

Определение 26. ***Алгоритм Монте-Карло*** *– такой рандомизированный алгоритм, что за детерминированное время выдаёт решение, являющееся корректным с вероятностью* $p > 0$. *Иными словами, алгоритм способен ошибаться.*

Решим задачу $Poly - Equal$ самым интуитивным способом – случайно выберем некоторое число x. Если многочлены принимают одинаковые значения на данном x, то будем считать их равными.

Алгоритм 14 Poly-Equal

Ввод: $A(x)$, $B(x)$, d

1: $x = randint(1,\ 100 \cdot d)$ – выбираем случайное число
2: **return** $A(x) = B(x)$

Очевидно, что алгоритм работает за полиномиальное время. Однако, ясно, что он может ошабаться. Оценим вероятность ошибки $\mathcal{P}$.

Пусть $A(x) \not\equiv B(x)$. Тогда ясно, что уравнение $A(x) - B(x) = 0$ будет иметь не более чем d корней (степени исходных ведь были d). Наш алгоритм ошибается, если x – один из корней данного уравнения (ведь значения многочленов на таких x совпадают). Тогда:

$$\mathcal{P}(\{\text{ошибка}\}) = \mathcal{P}(\{\text{выбран корень } A(x) - B(x) = 0\}) \leqslant \frac{d}{100 \cdot d} = \frac{1}{100}$$

– то есть, мы ошибаемся менее чем в 1% случаев.

Ясно, что данный алгоритм является алгоритмом Монте-Карло.

Попробуем уменьшить вероятность ошибки. Сделаем это самым простым способом: выбираем не одно x, а k различных случайных x, на которых будем прогонять алгоритм.

Алгоритм 15 Poly-Equal

Ввод: $A(x)$, $B(x)$, d, k

1: $equal =$ **True** – индикатор ошибки
2: **for** $i = 1$ **to** k **do**
3: $\quad x = randint(1,\ 100 \cdot d)$ – выбираем случайное число
4: $\quad equal = equal \wedge \big(A(x) = B(x)\big)$ – логическое **и**
5: **return** $equal$

Ясно, что если $A(x) \equiv B(x)$, то $\mathcal{P}(\{\text{ошибка}\}) = 0$. В то же время, если $A(x) \not\equiv B(x)$, то:

$$\mathcal{P}(\{\text{ошибка}\}) = \mathcal{P}(\{\text{выбран корень } A(x) - B(x) = 0\}) \leqslant \left(\frac{d}{100 \cdot d}\right)^k = \left(\frac{1}{100}\right)^k$$

Таким образом, алгоритм остался полиномиальным, но вероятность ошибки уменьшилась экспоненциально, что очень даже хорошо.

Данный алгоритм также является алгоритмом Монте-Карло.

Алгоритм Лас-Вегас

Определение 27. ***Алгоритм Лас-Вегас*** *– такой рандомизированный алгоритм, что всегда даёт корректное решение поставленной задачи, однако, работает за случайное время. Иными словами, время работы данного алгоритма – некая случайная величина, и в одном случае алгоритм может дать быстрое полиномиальное решение, а в другом проработает экспоненциальное от входа время.*

Задачу $Poly - Equal$ можно решить так, чтобы мы всегда получали верный ответ. У уравнения $A(x) - B(x) = 0$ не более d корней, и предыдущие алгоритмы с пусть и очень маленькой, но некоторой вероятностью могли натыкаться на данные корни постоянно, тем самым выдавая неправильный ответ. Заметим, что если выбрать $d+1$ различное число, то тогда хотя бы одно среди данных чисел не будет являться корнем уравнения. Этот факт и будет лежать в основе нашего алгоритма.

Алгоритм 16 Poly-Equal

Ввод: $A(x)$, $B(x)$, d

```
order = sample(range(1, 100 · d + 1), d + 1) выбираем d + 1 случайное число
for x ∈ order do
    if A(x) ≠ B(x) then
        return False
return True
```

Ясно, что среди выбранных x-ов хотя бы один да и не будет корнем $A(x) - B(x) = 0$. Тогда, если многочлены не совпадают, то $A(x) - B(x) \neq 0$ и мы получим ответ **False**. В то же время, если и на данном x, не являющемся корнем $A(x) - B(x) = 0$, значения многочленов совпали, то многочлены равны (мы всегда получаем верное равенство) – **True**.

Ясно, что если многочлены совпали, то алгоритм отработает все $d+1$ итерацию по выбранным числам. В то же время, не известно, сколько именно времени работает наш алгоритм, если многочлены не совпадают. Для этого можно сравнить алгоритм с предыдущим алгоритмом Монте-Карло, взяв за основу k итераций.

Обозначим как A_i – событие $\{A(order[i]) = B(order[i])\}$. Тогда:

$$\mathcal{P}(\geqslant k\{\text{ итераций}\}) = \mathcal{P}\left(\bigcap_{i=1}^{k} A_i\right) = \frac{d}{100 \cdot d} \cdot \frac{d-1}{100 \cdot d - 1} \cdot \ldots \cdot \frac{d-k+1}{100 \cdot d - k + 1} \leqslant \left(\frac{d}{100 \cdot d}\right)^k = \left(\frac{1}{100}\right)^k$$

То есть, в случае несовпадения многочленов вероятность работать долго мала.

Приближённое решение задачи $3SAT$. Задача $Max - 3SAT$

Рандомизированный алгоритм

Рассмотрим NP-полную задачу $3SAT$. Формально, полиномиального алгоритма решения данной задачи не существует. Будем искать приближённое решение, а именно – такое означива-

ние, при котором число истинных конъюнктов (скобок с дизъюнцкциями) будет максимально. Данная задача называется **Задачей** $Max-3SAT$.

Для начала, найдём математическое ожидание числа истинных скобок. Пусть у нас есть n переменных $x_1, \ldots, x_n$ и m скобок. Каждая переменная x_i с вероятностью $p = \frac{1}{2}$ будет принимать значения 1 или 0.

$$\varphi = (l_1^1 \vee l_2^1 \vee l_3^1) \wedge \ldots \wedge (l_1^m \vee l_2^m \vee l_3^m)$$

Положим T – случайная величина, отвечающая числу истинных скобок. Пусть A_q – событие $\{q$-я скобка истинна$\}$. Тогда можно ввести индикатор события I_q: $I_q = \begin{cases} 1, \text{ если } A_q \text{ произошло} \\ 0 \text{ иначе} \end{cases}$.

В таком случае

$$T = I_1 + I_2 + \ldots + I_m$$

откуда

$$\mathsf{E}(T) = \mathsf{E}\big(I_1 + I_2 + \ldots + I_m\big) = \sum_{q=1}^{m} \mathsf{E}I_q$$

Из теории вероятности мы знаем, что $\mathsf{E}(I_q) = \mathcal{P}(A_q)$. Посчитаем вероятность того, что q-я скобка истинна. Ясно, что дизъюнкция из 3-х литер ложна если и только если ложны все дизъюнкты. Каждая литера принимает значения 0 и 1 с одинаковой вероятностью $p = \frac{1}{2}$. Тогда вероятность того, что q-я скобка ложна, равна $p^3 = \frac{1}{8}$, откуда $\mathcal{P}(A_q) = 1 - p^3 = \frac{7}{8}$. Но тогда:

$$\mathsf{E}(T) = \sum_{q=1}^{m} \mathsf{E}(I_q) = \sum_{q=1}^{m} \mathcal{P}(A_q) = \frac{7}{8}m$$

– то есть, для случайного означивания мы в среднем получим $\frac{7}{8}$ от оптимального решения.

Отсюда следует, что для любой $3SAT$-формулы φ существует означивание, делающее хотя бы $\frac{7}{8}$ всех скобок истинными. Отсюда алгоритм, максимизирующий число истинных скобок, даст означивание, при котором не менее $\frac{7}{8}$ истинных скобок. Следовательно, любое решение задачи $Max-3SAT$ для формулы φ является означиванием, при котором истинны не менее $\frac{7}{8}m$ скобок.

Теперь будем решать задачу $Max-3SAT$. Она NP-трудна, так как иначе для выполнимой 3КНФ мы бы смогли найти максимальное означивание (то есть по факту выполняющее), решив задачу $Max-3SAT$, и потом подставить в $3SAT$. Тогда $3SAT \in \mathsf{P}$, что формально не является правдой.

Договоримся, что во всех скобках литеры отвечают разным переменным. Будем перебирать случайные означивания до тех пор, пока не найдём такое, при котором хотя бы $\frac{7}{8}$ всех скобок истинны

Алгоритм 17 Randomized-Max-3SAT

Ввод: φ – 3КНФ, n

1: **repeat**
2: $a =$ **random_assignment**(n) – случайное означивание n переменных, каждая истинна с вероятностью $\frac{1}{2}$
3: **until** $\varphi(a)$ содержит $\geqslant \frac{7}{8}m$ истинных скобок
4: **return** a

Как показано выше, такое означивание существует, значит, мы его точно найдём. Случайной величиной тут является время работы алгоритма (сколько придётся перебрать означиваний, прежде чем найдём нужное), значит, можем сделать вывод, что **Randomized-Max-3SAT** является алгоритмом Лас-Вегас.

Посчитаем математическое ожидание числа итераций, чтобы оценить время работы.

Пусть T_j – событие {при данном означивании ровно j скобок истинны}. Обозначим как P вероятность события {истинны $\geqslant \frac{7}{8}m$ скобок}. Тогда ясно, что

$$P = \sum_{j \geqslant \frac{7}{8}m}^{m} \mathcal{P}(T_j)$$

и матожиданием времени работы (числа итераций) алгоритма будет число $\frac{1}{P}$.

Выразим P через известное матожидание числа истинных скобок:

$$\frac{7}{8}m = \mathsf{E}(T) = \sum_{j=0}^{m} j \cdot \mathcal{P}(T_j) = \sum_{j=0}^{m' < \frac{7}{8}m} j \cdot \mathcal{P}(T_j) + \sum_{j \geqslant \frac{7}{8}m}^{m} j \cdot \mathcal{P}(T_j) \leqslant$$

$$\Downarrow \left\{ m' = \left\lceil \frac{7}{8}m \right\rceil - 1 \right\} \Downarrow$$

$$\leqslant m' \cdot \sum_{j=0}^{m'} \mathcal{P}(T_j) + m \cdot \sum_{j \geqslant \frac{7}{8}m}^{m} \mathcal{P}(T_j) = m' \cdot (1 - P) + m \cdot P \leqslant m' + m \cdot P$$

$$\Downarrow$$

$$m \cdot P \geqslant \frac{7}{8}m - m' \geqslant \frac{1}{8} \Longrightarrow P \geqslant \frac{1}{8 \cdot m}$$

Последняя оценка объясняется тем, что если $\frac{7}{8}m$ не целое число, то разность даст нам число вида «целая часть и сколько-то восьмых». Значит, данное число равно хотя бы $\frac{1}{8}$.

Таким образом, матожидание числа итераций алгоритма равно:

$$\text{время в среднем} = \frac{1}{P} = \left\{ P = \frac{1}{8 \cdot m} \right\} = 8 \cdot m$$

Что в среднем даёт хорошее время работы.

Детерминированный алгоритм

Существует также приближённое решение задачи $Max - 3SAT$, работающее за фиксированное время. Идея такова: будем брать по одной переменной x_i и смотреть два варианта означивания: $x_i = 1$ и $x_i = 0$. Затем рассматриваем всю 3КНФ φ как булеву формулу от уменьшенного на 1 числа переменных с фиксированными значениями $x_i = 1$ – в этом случае возникнет 3КНФ φ_t – и $x_i = 0$ – в этом случае возникнет 3КНФ φ_f.

Покажем, что такое решение корректно. Для этого достаточно доказать, что сохраняется инвариант на ограничение математического ожидания числа истинных скобок. То есть, если $T(\varphi)$ – число истинных скобок в исходной булевой формуле φ, а $T(\varphi_t)$ и $T(\varphi_f)$ – в формулах φ_t и φ_f соответственно, то

$$\mathsf{E}\big(T(\varphi)\big) \geqslant \frac{7}{8}m \Longrightarrow \left[\begin{array}{l} \mathsf{E}\big(T(\varphi_t)\big) \geqslant \frac{7}{8}m \\ \mathsf{E}\big(T(\varphi_f)\big) \geqslant \frac{7}{8}m \end{array} \right.$$

Распишем матожидание по линейности и сразу же получим требуемое:

$$\mathsf{E}\big(T(\varphi)\big) = \mathsf{E}T(\varphi_t)\cdot\mathcal{P}(\{x_i=1\}) + \mathsf{E}T(\varphi_f)\cdot\mathcal{P}(\{x_i=0\}) = \frac{1}{2}\Big(ET(\varphi_t) + \mathsf{E}T(\varphi_f)\Big)$$

$$\Downarrow \mathsf{E}\big(T(\varphi)\big) \geqslant \frac{7}{8}m \Downarrow$$

$$\frac{1}{2}\mathsf{E}T(\varphi_t) + \frac{1}{2}\mathsf{E}T(\varphi_f) \geqslant \frac{7}{8}m$$

$$\Downarrow \text{так как одновременно оба точно не могут быть меньше} \Downarrow$$

$$\left[\begin{array}{l} \mathsf{E}\big(T(\varphi_t)\big) \geqslant \frac{7}{8}m \\ \mathsf{E}\big(T(\varphi_f)\big) \geqslant \frac{7}{8}m \end{array}\right.$$

Значит, инвариант сохраняется. По окончании работы алгоритма неозначенных переменных вообще не останется, а матожидание числа истинных скобок $\mathsf{E}\big(T(\varphi)\big) \geqslant \frac{7}{8}m$. Значит, мы получим означивание– решение $Max-3SAT$, и наш алгоритм корректен.

Итоговый алгоритм будет иметь детерминированное время работы, и выглядит так:

Алгоритм 18 Deterministic-Max-3SAT

Ввод: φ – 3КНФ, n

1: **for** $i = 1$ **to** n **do**
2: $\quad\varphi_t = \varphi[x_i = \textbf{True}]$ – означивание с $x_i = 1$
3: $\quad\varphi_f = \varphi[x_i = \textbf{False}]$ – означивание с $x_i = 0$
4: $\quad$**if** $\mathsf{E}\big(T(\varphi_t)\big) \geqslant \mathsf{E}\big(T(\varphi_f)\big)$ **then**
5: $\quad\quad x_i = \textbf{True}$
6: $\quad\quad\varphi = \varphi_t$ – переходим к меньшей КНФ
7: $\quad$**else**
8: $\quad\quad x_i = \textbf{False}$
9: $\quad\quad\varphi = \varphi_f$ – переходим к меньшей КНФ
10: **return** $(x_1,\, x_2,\, \ldots,\, x_n)$

Потоковые алгоритмы. Задача о частотах.

Детерминированное решение

Пускай у нас есть поток S из n элементов: $t_1, t_2, \ldots, t_n$.

Определение 28. ***Частота*** *элемента t в потоке S – число f_t, равное количеству индексов i элементов потока, таких, что $t_i = t$.*

Рассмотрим **Задачу о частотах**, которую мы хотим решить для потока S:

Frequency

Ввод: $S = \{t_1, t_2, \ldots, t_n\}$ – поток

Вывод: Частоты f_t элементов потока S, а также самые частые элементы потока S, а именно такие элементы t, что $f_t \geqslant \varepsilon \cdot n$ для некоторого $\varepsilon > 0$.

Для маленьких размеров потока можно просто считать поток и проитерироваться по нему, считая частоты. Однако, если объём поступающих данных превышает максимальную вместимость по памяти у нашей вычислительной машины, то данное решение не оптимально. Рассмотрим пример.

Имеется поток S следующего вида:

$$S = \underbrace{t_1, t_2, t_3, \ldots, t_n}_{\text{все элементы различны}}, \underbrace{t_{n+1}, t_{n+2}, \ldots, t_{2n-1}}_{n-1 \text{ одинаковых элементов}}$$

Возьмём $\varepsilon = \dfrac{1}{2}$, в таком случае нужно найти элемент, встречающийся более 50%. Ясно, что это элемент $t = t_{n+1}$. Однако, чтобы понять, что $f_t > 50\%$, нужно найти этот элемент среди первых $t_1, \ldots, t_n$, иначе будет лишь $n-1$ повторений, а $\dfrac{n-1}{2n-1} < 50\%$. Следовательно, мы не сможем обойтись памятью меньшей, чем $O(n)$. Значит, такой банальный способ решения данной задачинам не подходит.

Будем решать задачу приближённо. Идея алгоритма такова: храним счётчики не для всех возможных различных элементов, а только фиксированное k штук, каждый из которых считает вхождение своего элемента. Стоит отметить, что $k \ll n$. Будем хранить счётчики в хэштаблице – ассоциативном массиве, который по ключу t выдаёт количество повторений элемента t. Ясно, что поскольку счётчиков много меньше, чем самих элементов, то в какой-то момент нам может попасться такой элемент t, что отличен от всех остальных k элементов, для которых мы завели счётчики. В этом случае провернём такой трюк: возьмём счётчик элемента s с самым маленьким значением и «заменим владельца» – полагаем с этого момента, что счётчик s считает элементы t, и что уже посчитанное число повторений элемента s является числом повторений элемента t. Таким образом, у нас всегда будет k счётчиков, просто у некоторых могут меняться владельцы.

Рассмотрим работу алгоритма на примере. Пусть $S = \{2, 3, 1, 5, 1, 1, 3\}$ и $k = 2$. Имеем хэштаблицу $T = [\,]$. Будем обозначать считанный элемент x как $\underline{x}$.

$$\underline{2} \Rightarrow T[2] = 1$$
$$\underline{3} \Rightarrow T[2] = 1, T[3] = 1$$
$$\underline{1} \Rightarrow T[2] \to T[1] + 1 \Rightarrow T[1] = 2, T[3] = 1$$
$$\underline{5} \Rightarrow T[3] \to T[5] + 1 \Rightarrow T[5] = 2, T[1] = 2$$
$$\underline{1} \Rightarrow T[1] = T[1] + 1 \Rightarrow T[1] = 3, T[5] = 2$$
$$\underline{1} \Rightarrow T[1] = T[1] + 1 \Rightarrow T[1] = 4, T[5] = 2$$
$$\underline{3} \Rightarrow T[5] \to T[3] + 1 \Rightarrow T[3] = 3, T[1] = 4$$
итоговые частоты: $T[1] = 4, T[3] = 3$

В итоге мы пусть и ошибёмся в значениях частот, но зато найдём самые частые элементы – 1 и 3. Однако, заметим, что если бы в потоке s был ещё один элемент, скажем, 6, то тогда:

$$\underline{6} \Rightarrow T[3] \to T[6] + 1 \Rightarrow T[6] = 4,\ T[1] = 4$$
итоговые частоты: $T[1] = 4,\ T[6] = 4$

Это значительно хуже, ведь элемент 6 один из самых редких.

Запишем теперь сам алгоритм поиска частот:

Алгоритм 19 Calculate_Frequency

Ввод: $S = \{t_1, t_2, \ldots, t_n\}$ – поток

$T = [\,]$ – хэштаблица
for $i = 1$ **to** n **do**
    проверяем, есть ли i-й счётчик в T
    if $t_i \in T$ **then**
        $T[t_i] = T[t_i] + 1$ – увеличиваем значение счётчика
    else if $|T| \leqslant k$ **then**
        $T[t_i] = 1$ – встретили новый элемент, и при этом не все счётчики заняты
    else
        $s = \underset{s \in T}{\mathbf{argmin}}\ T[s]$ – находим элемент, дающий минимальное значение счётчика
        $T[t_i] = T[s] + 1$ – теперь счётчик s считает t_i
        delete(T, s) – удаляем старый счётчик
return T – возвращаем значения счётчиков

Оценим величину ошибки, а именно, насколько сильно мы ошиблись в подсчёте частот «якобы» самых частых элементов.

Пусть $\overset{\triangledown}{\mathbf{min}}$ – минимум среди всех счётчиков. Заметим, что его значение может быть не больше $\frac{n}{k}$:

$$\overset{\triangledown}{\mathbf{min}} = \min_{s \in T} T[s] \leqslant \frac{n}{k}$$

так как всего n элементов и k счётчиков, и все n элементов потока распределяются по k счётчикам, которые в сумме и дадут данное n.

Пусть f_t^i – это число, равное количеству раз, сколько мы встретили элемент t среди первых i элементов потока S. Пусть также T^i – состояние счётчиков (то бишь хэштаблицы) после i-й итерации алгоритма. Докажем следующую лемму:

Лемма.

- $$t \in T^i \Longrightarrow f_t^i \leqslant T^i[t]$$

  То есть, после i-й итерации счётчик элемента t посчитал количество повторений, не меньшее, чем истинное число повторений элемента t в потоке среди первых i.

- $$s \notin T^i \Longrightarrow f_i^s \leqslant \overset{\triangledown}{\boldsymbol{min}}^i$$

  То есть, если в данный момент мы не считаем элемент s, то частота его появления в потоке среди первых i элементов не больше, чем минимальное значение счётчика на i-м шаге.

Доказательство. Будем доказывать индукцией по числу итераций. База для $i = 0$ очевидна: все значения f_t^i, $\overset{\nabla}{\mathbf{min}}^i$, $T^i[t]$ равны 0.

Пусть для первых $i-1$ итераций алгоритма оба условия выполнены. Покажем, что всё выполнено и на i-й итерации.

Мы считали элемент t_i. Всего есть 3 случая, что могли произойти:

1. $t_i \in T^{i-1}$ – счётчик элемента уже есть в таблице: $t_i \in T^{i-1} \to t_i \in T^i$

   В данном случае для элемента t_i произошло увеличение частоты: $\begin{cases} f_{t_i}^i = f_{t_i}^{i-1} + 1 \\ T^i[t_i] = T^{i-1}[t_i] + 1 \end{cases}$

   Ясно, что что-либо поменялось только для частоты и счётчика t_i. Тогда, пользуясь предположением индукции:

   $$f_{t_i}^{i-1} \leqslant T^{i-1}[t] \Longrightarrow f_{t_i}^{i-1} + 1 \leqslant T^{i-1}[t_i] + 1 \iff f_{t_i}^i \leqslant T^i[t_i]$$

2. $|T^{i-1}| \leqslant k$ – счётчика элемента в таблице не было, но он теперь он появился и занял свободное место: $t_i \notin T^{i-1} \to t_i \in T^i$

   В данном случае мы встретили элемент t_i впервые, откуда $f_{t_i}^i = 0 + 1 = 1$. В таблице счётчик также возник впервые: $T^i[t_i] = 1$. Тогда очевидно $f_{t_i}^i \leqslant T^i[t_i]$.

3. **замена владельца** – счётчика элемента в таблице не было, но теперь он появился, заняв место счётчика другого элемента s: $\begin{cases} t_i \notin T^{i-1} \to t_i \in T^i \\ s \in T^{i-1} \to s \notin T^i \end{cases}$

   Поскольку на предыдущем, $i-1$-м шаге, элемента t_i в таблице не было, то по предположению индукции $f_{t_i}^{i-1} \leqslant \overset{\nabla}{\mathbf{min}}^{i-1}$. Заметим, что $\overset{\nabla}{\mathbf{min}}^{i-1} = T^{i-1}[s]$. Тогда:

   $$\begin{cases} f_{t_i}^i = f_{t_i}^{i-1} + 1 \\ T^i[t_i] = T^{i-1}[s] + 1 \end{cases} \Longrightarrow f_{t_i}^i \leqslant \overset{\nabla}{\mathbf{min}}^{i-1} + 1 = T^{i-1}[s] + 1 = T^i[t_i]$$

   Рассмотрим s. По предположению индукции, $f_s^{i-1} \leqslant \overset{\nabla}{\mathbf{min}}^{i-1}$. Но при этом $\overset{\nabla}{\mathbf{min}}^{i-1} = T^{i-1}[s]$. Ясно, что $f_s^i = f_s^{i-1}$ – счётчик пропадает из таблицы, и для него ничего не меняется. Однако, $\overset{\nabla}{\mathbf{min}}$ может увеличиться, а может и не измениться:

   $$\begin{bmatrix} \overset{\nabla}{\mathbf{min}}^i = \overset{\nabla}{\mathbf{min}}^{i-1} \\ \overset{\nabla}{\mathbf{min}}^i = \overset{\nabla}{\mathbf{min}}^{i-1} + 1 \end{bmatrix} \Longrightarrow \overset{\nabla}{\mathbf{min}}^{i-1} \leqslant \overset{\nabla}{\mathbf{min}}^i$$

   В любом случае, выполняется неравенство $f_s^i \leqslant \overset{\nabla}{\mathbf{min}}^i$.

Таким образом, в каждом из 3-х возможных случев оба неравенства выполняются и для i-го элемента потока. Шаг индукции доказан, а значит, и сама лемма тоже. □

Рассмотрим теперь финальное состояние таблицы T в котором алгоритм закончил работу. Из леммы выше следует, что для каждого элемента t потока, у которого есть счётчик в таблице, выполнено, что $f_t \leqslant T[t]$. То есть, все счётчики набрали не меньше истинной частоты соответствующих элементов. Оценим частоту t снизу.

Пусть $t \in T$. Посмотрим на тот момент, когда t оказался в таблице и больше оттуда не исчезал. Пускай это произошло на j-й итерации. Возможны следующие случаи:

- Мы завели новый счётчик, и $T^j[t] = 1$. Тогда $f_t = T[t]$ – мы посчитаем в точности число повторений элемента t.
- Произошла замена владельца в последний раз, то есть, после того, как t забрал счётчик у другого элемента s, его счётчик больше никто не отбирал до конца работы алгоритма.

Изучим второй случай подробнее. Поскольку на j-й итерации t забрал счётчик у s, то значит, на предыдущей итерации его в таблице не было, и $t \notin T^{j-1}$. Тогда по лемме выше:

$$f_t^{j-1} \leqslant \overset{\nabla}{\mathbf{min}}^{j-1} \leqslant \frac{n}{k}$$

На момент j-й итерации счётчик элемента s был минимальным, откуда $T^{j-1}[s] = \overset{\nabla}{\mathbf{min}}^{j-1} \leqslant \frac{n}{k}$. Значит, более чем на $\frac{n}{k}$ мы точно не ошибёмся. Однако, начиная с j-й итерации считается элемент t, а не s. Тогда можно оценить f_t снизу:

$$f_t^{j-1} \leqslant \overset{\nabla}{\mathbf{min}}^{j-1} = T^{j-1}[s] \leqslant \frac{n}{k} \Longrightarrow f_t \geqslant T[t] - \frac{n}{k}$$

Заметим, что если некоторый элемент $s \notin T$, то $f_s \leqslant \overset{\nabla}{\mathbf{min}} \leqslant \frac{n}{k}$. Значит, данный элемент точно не является частым. Тогда таблице среди k элементов будут содержаться два типа элементов – реально самые частые элементы потока t, для которых частота f_t связана соответствующим неравенством $f_t \geqslant \frac{n}{k}$, а также те элементы r, для которых значение счётчика получилось с ошибкой не более чем на $\frac{n}{k}$ повторений по сравнению с реальной частотой: $f_r \geqslant T[r] - \frac{n}{k}$. Таким образом, алгоритм приближённо решает задачу поиска самых частых элементов, гарантированно находя все реально самые частые, а для остальных получает значение частоты с ошибкой. Частота f_t каждого элемента t получается ограниченной:

$$T[t] - \frac{n}{k} \leqslant f_t \leqslant T[t]$$

Однако, мы хотим найти такие элементы t, что $f_t \geqslant \varepsilon \cdot n$ для некоторого $\varepsilon > 0$. Поэтому оценим константу k – число счётчиков, чтобы понять, сколько именно счётчиков нам нужно делать для необходимого по условию приближения.

Запишем теперь сам алгоритм, решающий **Задачи о частотах**, а именно – находящий приближённые частоты f_t всех элементов таблицы T, которая в свою очередь содержит k самых частых элементов.

Алгоритм 20 Frequency

Ввод: $S = \{t_1, t_2, \ldots, t_n\}$ – поток, t – элемент, частоту которого мы хотим проверить

1: $T =$ **Calculate_Frequency**(S) – считаем частоты при помощи алгоритма выше
2: **if** $t \in T$ **then**
3: **return** $T[t]$ – возвращаем значение счётчика частого элемента
4: **else**
5: **return** $\frac{n}{k}$ – возвращаем верхнюю оценку для нечастого элемента

Поскольку самые частые элементы – такие элементы t, что $f_t \geqslant \varepsilon \cdot n$ для некоторого $\varepsilon > 0$, нужно, чтобы алгоритм возвращал ответ требуемой точности, а именно:

$$\begin{cases} f_t \leqslant \mathbf{Frequency}(t) \\ f_t \geqslant \mathbf{Frequency}(t) - \varepsilon \cdot n \end{cases}$$

Тогда возьмём $k = \dfrac{1}{\varepsilon}$, и из оценки выше следует:

$$T[t] - \frac{n}{k} \leqslant f_t \leqslant T[t]$$
$$\Downarrow k = \frac{1}{\varepsilon} \Longrightarrow \frac{n}{k} = \frac{\varepsilon \cdot n}{1} = \varepsilon \cdot n \Downarrow$$
$$\textbf{Frequency}(t) - \varepsilon \cdot n \leqslant f_t \leqslant \textbf{Frequency}(t)$$
$$\Downarrow$$
$$f_t \geqslant \varepsilon \cdot n \text{ для реально самых частых и } f_t \geqslant \textbf{Frequency}(t) - \varepsilon \cdot n \text{ для нечастых}$$

Таким образом, при $k = \dfrac{1}{\varepsilon}$ наш алгоритм действительно находит самые частые элементы потока, причём с необходимой по условию точностью, и он возвращает корректное приближённое решение задач о частотах. При этом, однако, он также может вернуть значения частот не самых частых элементов, но при этом они также ограничены точной оценкой. Можно также отметить, что поскольку параметр точности $\varepsilon > 0$ может быть достаточно внушительным (близким к 1), то величина $k = \dfrac{1}{\varepsilon}$ мала и не зависит от n, значит, наш алгоритм эффективен по памяти, ибо хранит всегда одно и то же количество счётчиков k вне зависимости от размера потока S.

Рандомизированное решение

Теперь рассмотрим другое решение **Задачи о частотах** – рандомизированное.

Пускай у нас есть поток S из n элементов: $t_1,\ t_2,\ \ldots,\ t_n$. Будем считать, что все $t \in U$ – некое универсальное множество со всеми объектами того же типа.

Зафиксируем d различных хэшфункций $h_1,\ h_2,\ \ldots,\ h_d$ с областью значений W, $h_i : U \to W$ где $W = \{1,\ 2,\ \ldots,\ w\}$. Полагаем, что все хэшфункции h хэшируют элементы из U равновероятно, то есть:

$$\mathcal{P}(\{h(t) == 1\}) = \mathcal{P}(\{h(t) == 2\}) = \ldots = \mathcal{P}(\{h(t) == w\}) = \frac{1}{w}$$

Затем, создаём таблицу размера $d \times w$. В ячейках она хранит счётчики элементов.

Хэшфункции $\downarrow$ \ Значения $\rightarrow$	1	2	…	j	…	$w-1$	w
h_1			…		…		$+1$
h_2		$+1$	…		…		
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
h_i			…	$+1$	…		
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
h_{d-1}	$+1$		…		…		
h_d			…		…	$+1$	
Для элемента $t \rightarrow$	$h_{d-1}(t)$	$h_2(t)$	…	$h_i(t)$	…	$h_d(t)$	$h_1(t)$

Заполняем мы её так: последовательно вычисляем значения всех d хэшфункций на каждом элементе t – сначала $h_1(t_1)$, потом $h_2(t_1)$, и так далее до $h_d(t_1)$, потом аналогично для $t_2,\ t_3,\ \ldots$. Если на данном элементе t хэшфункция h_i выдала значение j, то увеличиваем на 1 значение счётчика $[i,\ j]$ в соответствующей ячейке таблицы $T\big[i,\ h_i(t)\big]$.

К примеру, выше приведён ход заполнения таблицы для элемента t при значениях хэшфункций: $h_1(t) = w$, $h_2(t) = 2,\ \ldots,\ h_i(t) = j,\ \ldots,\ h_{d-1}(t) = 1$, $h_d(t) = w - 1$.

Пускай нам необходимо выяснить, какова частота f_t элемента t. Заметим, что каждый раз, когда мы встречали данный элемент в потоке, мы прибавляли 1 к счётчику $T\big[i,\, h_i(t)\big]$, в который нас захешировала функция h_i. Однако, в каждую соответствующую ячейку для хэшфункций $h_1,\, h_2,\, \ldots,\, h_d$ мы могли также прибавить 1 от какого-то другого элемента, не являющегося t. Тогда ясно, что самым точным приближённым значением частоты f_t будет минимальное значение счётчика для элемента t среди всех d хэшфункций. Его мы будем обозначать как $c(t)$.

Можно теперь построить сам алгоритм:

Алгоритм 21 Randomized-Frequency

Ввод: $S = \{t_1,\, t_2,\, \ldots,\, t_n\}$ – поток, t – элемент, частоту которого мы хотим проверить

1: **for** $q = 1$ **to** n **do**
2: **for** $i = 1$ **to** d **do**
3: $T\big[i,\, h_i(t_q)\big] = T\big[i,\, h_i(t_q)\big] + 1$ – считаем частоты
4: **return** $c_t = \min\limits_{i=1\ \textbf{to}\ d} T\big[i,\, h_i(t)\big]$ – наиболее точное приближённое значение

Оценим вероятнось того, что наша частота c_t отличается от реальной частоты f_t не очень сильно. Пускай на у нас есть два параметра – ε и δ, которые определяют необходимую по условию точность приближённого решения – как ε для самых частых в предыдущей задаче. И так же, как и в предыдущей задаче с k, мы хотит подобрать такие значения количеств d хэшфункций и w значений хэшей, чтобы вероятность:

$$\mathcal{P}\big(c_t - f_f \geqslant \varepsilon \cdot n\big) \leqslant \delta$$

Рассмотрим i-ю хэшфункцию, и оценим сверху вероятность ошибки для соответствующего счётчика элемента t:

$$\mathcal{P}\big(T\big[i,\, h_i(t)\big] - f_f \geqslant \varepsilon \cdot n\big) \leqslant \text{по неравенству Маркова} \leqslant \frac{\mathsf{E}\big(T\big[i,\, h_i(t)\big] - f_f\big)}{\varepsilon \cdot n}$$

$T\big[i,\, h_i(t)\big] - f_t$ – число тех элементов, что не равны t, но попали в данный счётчик. Пусть I_q – индикатор события $\{h_i(t_q) = h_i(t)\}$. Тогда распишем матожидание по определению:

$$\frac{\mathsf{E}\big(T\big[i,\, h_i(t)\big] - f_f\big)}{\varepsilon \cdot n} = \frac{\sum\limits_{q=1}^{n} I_q \cdot \mathcal{P}\big(\{h_i(t_q) = h_i(t)\}\big)}{\varepsilon \cdot n} \leqslant$$

$$\Downarrow \Big\{\text{берём максимально возможное количество произошедших событий}\Big\} \Downarrow$$

$$\leqslant \frac{n \cdot \mathcal{P}\big(\{h_i(t_q) = h_i(t)\}\big)}{\varepsilon \cdot n} = \Big\{\mathcal{P}\big(\{h_i(t_q) = h_i(t)\}\big) = \frac{1}{w}\Big\} = \frac{n}{\varepsilon \cdot n \cdot w} = \frac{1}{\varepsilon \cdot w}$$

Тогда возьмём $w = \dfrac{e}{\varepsilon}$ и получим:

$$\mathcal{P}\big(T\big[i,\, h_i(t)\big] - f_f \geqslant \varepsilon \cdot n\big) \leqslant \frac{\varepsilon}{e \cdot \varepsilon} = \frac{1}{e}$$

Вернёмся теперь к исходной вероятности. Поскольку мы получили верхнюю оценку для какой-то одной хэшфункции, можем подставить данное значение вообще для всех. Тогда:

$$\mathcal{P}\big(c_t - f_f \geqslant \varepsilon \cdot n\big) \leqslant \Big\{\mathcal{P}\big(T\big[i,\, h_i(t)\big] - f_f \geqslant \varepsilon \cdot n\big) \leqslant \frac{1}{e}\Big\} \leqslant \left(\frac{1}{e}\right)^d$$

Тогда возьмём $d = \ln \dfrac{1}{\delta}$ и получим:

$$\mathcal{P}\big(c_t - f_f \geqslant \varepsilon \cdot n\big) \leqslant \left(\frac{1}{e}\right)^d = \left\{d = \ln \frac{1}{\delta}\right\} = \left(\frac{1}{e}\right)^{\ln \frac{1}{\delta}} = \delta$$

Значит, при $w = \dfrac{e}{\varepsilon}$ и $d = \ln \dfrac{1}{\delta}$ мы получаем необходимую точность решения, и при этом расходы по памяти составят лишь $\dfrac{e}{\varepsilon} \cdot \ln \dfrac{1}{\delta}$, что не зависит от размера входа.

Переборные задачи

Общие факты

Пускай у нас есть какая-то задача – поиск пути в графе, поиск выполняющего означивания и подобные. Мы хотим найти не одно какое-то решение одной из данных задач, а вообще все решения – все пути, все выполняющие означивания и так далее. Возникает проблема: размер выхода может быть экспоненциальным (если не больше) от размера входа. Поэтому обычная оценка времени работы и полиномиальности решения здесь ничего не даёт – все алгоритмы можно считать плохими в этом отношении. Однако, разбираться как-то да надо. Поэтому необходимы иные подходы и методы, позволяющие сравнивать эффективность решений переборных задач. Рассмотрим некоторые из них.

Определение 29. ***Тотальный подход*** *– оцениваем время работы от размера и входа, и выхода.*

Соответственно, решение называется ***тотально полиномиальным****, если оно занимает полиномиальное от входа и выхода время.*

Определение 30. ***Инкрементальная полиномиальная задержка*** *– оцениваем время работы от размера входа и суммарного объёма выданный решений, каждое из которых должно быть полиномиально от размера входа.*

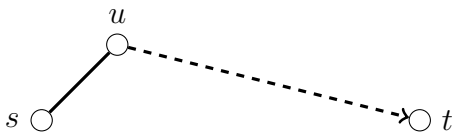
Соответственно, решение называется ***инкрементально полиномиальным****, если оно занимает полиномиальное от входа и суммарного размера выданных решений время.*

Определение 31. ***Полиномиальная задержка*** *– оцениваем время работы от размера входа и объёма вычислений, которые алгоритм производит между выдачей двух последовательных решений, каждое из которых должно быть полиномиально от размера входа.*

Соответственно, решение называется ***полиномиальным с задержкой****, если между выдачей двух последовательных полиномиальных решений тратится также полином.*

Все пути между вершинами

Возьмём в графе $G = (V, E)$ две вершины – s и t. Мы хотим найти все пути, ведущие из вершины s в вершину t.



Пусть в графе есть ребро (s, u). Ясно, что путь из s в t либо начинается с данного ребра, либо нет. Тогда будем действовать по принципу «разделяй и властвуй» – запустим алгоритм поиска всех путей от вершины u к вершине t, решив аналогичную задачу для подграфа без вершины s. Далее, поскольку мы уже изучили ребро (s, u), выкинем его из графа, и решим аналогичную задачу для подграфа без ребра (s, u). Таким образом мы сможем действительно получить вообще все возможные пути из s в t.

К сожалению, в такой реализации мы не можем даже точно оценить время работы алгоритма – ведь при рассмотрении новых рёбер или вершин мы можем блуждать по графу, перебирая пути, невероятно долго, а при этом вершина t может и вовсе быть недостижимой из данной u, и нет смысла перебирать все варианты несуществующих путей с ребром (s, u).

Будем тогда перед запуском рекурсивного алгоритма проверять вершину t на достижимость из текущей позиции (при помощи того же поиска в глубину). Это позволит нам избежать уймы лишних действий, и алгоритм будет иметь такой вид:

Алгоритм 22 All_Path

Ввод: $G = (V, E)$, s, t, P где P – текущий путь

1: **if** $s = t$ **then**
2: **print** P – выводим путь
3: **return**
4: выбираем ребро $(s, u) \in E$
5: **if** $\exists\ u \to t$ – путь в подграфе без s **then**
6: **All_Path**$(G \setminus \{s\}, u, t, P \cup \{(s, u)\})$
7: **if** $\exists\ s \to t$ – путь в подграфе без (s, u) **then**
8: **All_Path**$(G \setminus \{(s, u)\}, u, t, P)$

В итоге мы получим эдакое дерево путей. В нём листьями будут все пути $s \to t$, высота этого дерева будет не больше $|V|$. Стоит отметить, что наш алгоритм является полиномиальным с задержкой, так как от листа к листу мы идём за полиномиальное время, проверяя на достижимость и добавляя рёбра к пути не более чем $|V|$ раз в каждом случае.

Максимальные по вложению клики

Представим, что у нас есть социальная сеть (какой-нибудь банальный ВКонтакте). В ней люди связаны множеством отношений – кто чей друг, кто в каких сообществах состоит и подобное. По факту, каждая социальная сеть являет собой граф, в котором люди – вершины, а отношения между людьми – рёбра.

Представим такую задачу. У нас имеется множество людей – вершин. Два человека соединены ребром, если они являются сослуживцами – работают в одной компании. Тогда всех людей можно разделить на непересекающиеся множества сослуживцев. Ясно, что если два человека имеют одного общего сослуживца, то они являются сослуживцами и для друг друга. Получается, что каждое такое сообщество сослуживцев является кликой – полным подграфом. Мы хотим найти все сообщества сослуживцев на заданном множестве людей.

Мы знаем, что **Задача о максимальной клике** NP-трудна. Будем вместо этого искать **максимальные по вложению** клики, то есть такие полные подграфы, что мы больше не можем добавить ни одну вершину так, чтобы подграф остался кликой.

Max-Attachment-Clique

Ввод: $G = (V, E)$ – неориентированный невзвешенный граф

Вывод: $C_1, \ldots, C_k$ – множества вершин графа, такие, что для каждого множества C_i выполнено $\begin{cases} \forall u \in C_i \ni v : (u, v) \in E \\ \forall\ w \notin C_i\ \exists\ u \in C_i \mid (w, u) \notin E \end{cases}$

Алгоритм поиска одной максимальной по вложению клики интуитивен и прост:

Алгоритм 23 Single-Max-Attachment-Clique

Ввод: $G = (V, E)$ – неориентированный невзвешенный граф

$V = \{v_1, v_2, \ldots, v_n\}$ – нумеруем множество вершин
$C = \{v_1\}$ – формируем клику на основе первой вершины
for $i = 2$ **to** n **do**
  if $C \cup \{v_i\}$ – клика **then**
    $C = C \cup \{v_i\}$ – увеличиваем клику
return C

Будем последовательно перебирать вершины и добавлять в клику те, что не нарушают полноты нашего подграфа-клики. По завершении мы получим множество вершин, такое, что любые две вершины из множества соединены ребром. Однако, необходимо найти вообще все максимальные по вложению клики. В этом случае алгоритм будет гораздо более сложным.

Занумеруем вершины нашего графа. Идея решения заключается в том, что мы формируем двоичное дерево, вершинами которого являются сами клики, причём на j-м уровне дерева находятся все максимальные клики, состоящие из первых j вершин графа. Начинать работу мы будем также с первой вершины v_1, а само дерево будет формироваться по особому правилу.

Предварительно введём на множестве всевозможных клик нашего графа **лексикографический порядок** следующим образом:

$$a_1 a_2 \dots a_n \prec b_1 b_2 \dots b_n \quad \Longleftrightarrow \quad \exists\, k:\ a_1 = b_1,\ a_2 = b_2,\ \dots,\ a_{k-1} = b_{k-1},\ a_k < b_k$$

Например, $1357 \prec 2346$. Договоримся, что сначала к рассмотрению идут лексикографически меньшие клики – тем самым мы будем строить наше дерево «в глубину», подобно обходу **DFS**.

Пусть на j-м уровне мы имеем клику C – максимальную клику из j вершин. Мы хотим, перейдя на $j+1$-й уровень, получить все максимальные клики на $j+1$ вершинах путём добавления вершины v_{j+1}. Могло возникнуть два случая:

1. $C \cup \{v_{j+1}\}$ – клика

   В этом случае мы добавляем нашей вершине, отвечающей клике C, потомка $C \cup \{v_{j+1}\}$, просто добавляя вершину v_{j+1} в клику. C была максимальной по включению на j-м уровне, значит, новая клика будет максимальной на $j+1$-м.

2. $C \cup \{v_{j+1}\}$ – не клика

   Иначе говоря, после добавления вершины v_{j+1} в клику C данный подграф перестанет быть полным. Ясно, что если клика C была максимальной j-го уровня, и мы не смогли добавить v_{j+1}, то клика C будет также максимальной и $j+1$-го уровня. Поэтому потомок C точно добавляется в дерево.

   Однако, вполне возможно, что v_{j+1} входит в максимальную клику $j+1$-го уровня. Тогда мы «проверяем» этот факт: выкидываем из клики C все вершины кроме тех, что связаны с v_{j+1} ребром – фактически мы получаем пересечение множества C и множества $N(v_{j+1})$ связных с v_{j+1} вершин.

   Ясно, что тогда образуется клика – множество $\{v \in C \mid (v,\, v_{j+1}) \in E\} \cup \{v_{j+1}\}$. НО: она может и клика, но не факт, что максимальная по вложению для $j+1$-го уровня. Более того, подобные клики – пересечение того, что было, с v_{j+1} – могли возникнуть сразу в нескольких местах нашего дерева – ведь клики формируются из $j+1$ первых вершин. Поэтому, прежде чем добавить $\big(C \cap N(v_{j+1})\big) \cup \{v_{j+1}\}$, необходимо проверять дополнительные условия.

Будем называть **каноническим предком** клики C $j+1$-го уровня такую клику C^*, что является лексикографически первым предком j-го уровня для нашей клики C. Он находится раньше всех остальных предков на пути обхода DFS. Алгоритм поиска канонического предка:

Алгоритм 24 Canonical Ancestor

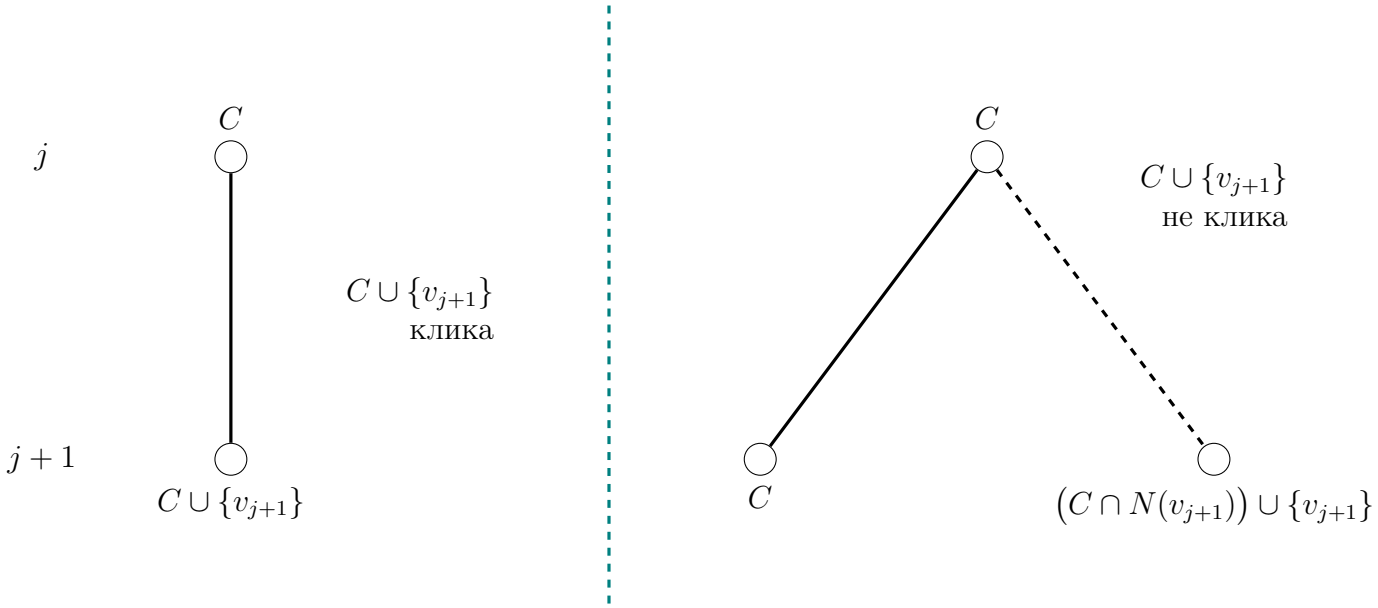
Ввод: C – клика, $j+1$ – уровень клики

```
P = C \ N(v_{j+1}) – клика без связных с v_{j+1} вершин
for i = 1 to j do
    if P ∪ {v_i} – клика then
        P = P ∪ {v_i} – добавляем вершины в лексикографической очерёдности
return P – канонический предок клики C
```

Канонический предок нам нужен для следующего: у каждой вершины вида $\big(C \cap N(v_{j+1})\big) \cup \{v_{j+1}\}$ существует такой предок – клика C^* – из которой данная клика получилась впервые при построении дерева. Тогда, если $\big(C \cap N(v_{j+1})\big) \cup \{v_{j+1}\}$ – максимальная по вложению клика $j+1$-го уровня, то добавляем в качестве потомка той и только той вершине C, что $C = C^*$ – в таком случае у нас не будет повторяющихся клик в дереве, и нам не придётся идти несколько раз по одним и тем же поддеревьям.

Можно сформулировать итоговое правило, по которому формируется дерево, а именно – как строятся потомки следующего уровня. Пусть у нас есть клика C j-го уровня. Рассматриваем $j+1$-ю вершину v_{j+1}. Тогда:

- Если $C \cup \{v_{j+1}\}$ – клика, то добавляем клике C потомка $C \cup \{v_{j+1}\}$ – максимальную клику $j+1$-го уровня.
- Если $C \cup \{v_{j+1}\}$ – не клика, то:
  1. Добавляем левого потомка – клику C, которая максимальная и $j+1$-го уровня тоже.
  2. Если клика $\big(C \cap N(v_{j+1})\big) \cup \{v_{j+1}\}$ – максимальная по вложению клика $j+1$-го уровня **и если** её канонический предок – это клика C, то добавляем правого потомка – клику $\big(C \cap N(v_{j+1})\big) \cup \{v_{j+1}\}$.



Докажем корректность такого подхода.

Лемма. *При построении дерева по вышеуказанному правилу сохраняется инвариант: на каждом j-м уровне дерева все вершины – максимальные клики j-го уровня.*

Доказательство. Будем доказывать по индукции. База очевидна – самая первая вершина v_1 является максимальной кликой 1-го уровня, так как больше и добавлять то нечего.

Теперь докажем шаг. Пусть C – максимальная клика $j+1$-го уровня. Покажем, что она получилась по правилу из некоторых клик j-го уровня. Могут возникнуть такие случаи:

- $v_{j+1} \in C$

  Пусть $C_0 = C \setminus \{v_{j+1}\}$ – клика без вершины v_{j+1}. По предположению индукции, она либо максимальная j-го уровня, либо содержится в таковой. Возьмём C^* – канонический предок C, то есть максимальная клика j-го уровня, такая, что $C_0 \subseteq C^*$.

  Тогда есть 2 варианта, как C^* соотносится с v_{j+1}:

1. $C^* \cup \{v_{j+1}\}$ – клика. Заметим, что она состоит из вершин клики C_0, каких-то ещё вершин и вершины v_{j+1}. Получается, что $C_0 \cup \{v_{j+1}\}$ – подклика в $C^* \cup \{v_{j+1}\}$. Но $C_0 \cup \{v_{j+1}\} = C \setminus \{v_{j+1}\} \cup \{v_{j+1}\} = C$. Следовательно, клики $C^* \cup \{v_{j+1}\}$ и C должны совпадать, так как в противном случае C – клика $j+1$-го уровня, являющаяся подкликой другой клики $j+1$-го уровня, а значит, не может быть максимальной – противоречие. Тогда $C = C^* \cup \{v_{j+1}\}$ – максимальная клика $j+1$-го уровня, полученная по правилу.

2. $C^* \cup \{v_{j+1}\}$ – не клика. Тогда возьмём $\overset{\nabla}{C} = \{v \in C^* \mid (v,\, v_{j+1} \in E\}$ – клику из всех связных с v_{j+1} вершин C^*. Ясно, что $C_0 \subseteq \overset{\nabla}{C}$, так как $C = C_0 \cup \{v_{j+1}\}$, а значит, все вершины C_0 связны с v_{j+1}.

   Предположим, что существует такая вершина u, что $u \in \overset{\nabla}{C} \setminus C_0$. $\overset{\nabla}{C} \cup \{v_{j+1}\}$ – клика $j+1$-го уровня. Но тогда

   $$C = C_0 \cup \{v_{j+1}\} \subseteq \overset{\nabla}{C} \cup \{v_{j+1}\}$$

   – C является подкликой другой клики $j+1$-го уровня, а значит, не максимальная – противоречие. Следовательно, таких вершин u не существует, откуда $\overset{\nabla}{C} = C_0$, и $C = \overset{\nabla}{C} \cup \{v_{j+1}\}$ – максимальная клика $j+1$-го уровня, полученная по правилу.

- $v_{j+1} \notin C$

  Но тогда, поскольку новой вершины v_{j+1} нет в C, то C не изменилась с предыдущего уровня и по предположению индукции является максимальная кликой и j-го уровня тоже. Значит, C – потомок той же C предыдущего уровня, и была получена по правилу.

Получается, что на $j+1$-м уровне все клики – максимальные, полученные по правилу из максимальных по вложению клик j-го уровня. Шаг доказан, значит, инвариант сохраняется. □

На основании данной леммы можно сделать вывод, что по завершении работы алгоритма листьями построенного дерева будут все максимальные по вложению клики исходного графа. Значит, он корректен. Высота этого дерева будет не больше $|V|$.

Стоит отметить, что в такой реализации наш алгоритм поиска всех максимальных по вложению клик в графе является полиномиальным с задержкой, так как переход между решениями можно гарантированно осуществить за полином от размера графа – при помощи того же **DFS** на построенном дереве клик.

Тогда, используя алгоритм поиска канонического предка:

Алгоритм 25 Canonical Ancestor

Ввод: C – клика, $j+1$ – уровень клики

$P = C \setminus N(v_{j+1})$ – клика без связных с v_{j+1} вершин
for $i = 1$ **to** j **do**
  if $P \cup \{v_i\}$ – клика **then**
    $P = P \cup \{v_i\}$ – добавляем вершины в лексикографической очерёдности
return P – канонический предок клики C

Можем построить алгоритм решения поставленной задачи, находящий все максимальные по вложению клики графа $G = (V,\, E)$. Он приведён ниже.

Алгоритм 26 Max-Attachment-Clique

Ввод: $G = (V, E)$ – неориентированный невзвешенный граф

```
function BUILD_TREE(C – клика, j – уровень)
    if j = |V| then
        print C – выводим максимальную клику
        return
    if ∃ v ∈ C | (v, v_{j+1}) ∉ E then
        C.left = C – левый потомок
        Build_Tree(C.left, j + 1) – строим дерево в глубину
        C_0 = {v_{j+1}} – множество из v_{j+1} и связных с ней вершин
        for v in C do
            if (v, v_{j+1}) ∈ E then
                C_0 = C_0 ∪ {v}
        is_Max = True – индикатор максимальности клики
        for i = 1 to j do
            if (v_i, v_{j+1}) ∈ E and v_i ∉ C_0 then
                is_Max = False – клика не будет максимальной
                break
        if is_Max and Canonical Ancestor(C_0, j + 1) = C then
            C.right = C_0 – правый потомок
            Build_Tree(C.right, j + 1) – строим дерево в глубину
    else
        C.left = C ∪ {v_{j+1}} – единственный потомок
        Build_Tree(C.left, j + 1) – строим дерево в глубину

V = {v_1, v_2, ..., v_n} – нумеруем множество вершин
C = {v_1} – формируем клику на основе первой вершины
return Build_Tree(C, 1)
```