Алгоритмически неразрешимые задачи. Алгоритмы построения минимальных остовных деревьев.

Поздняков Сергей Николаевич

запись конспекта: Ковалева Ксения
дата лекции: 20.12.2018

0:00

Большая теорема Ферма.

На предыдущем занятии мы обсуждали проблему вычислимости и выяснили, что не для каждой задачи существует алгоритм, который её решает.

В качестве простого соображения мы привели пример того, что количество языков, которые мы можем построить на каком-то конечном алфавите, эквивалентно числу подмножеств множества символов алфавита. Но поскольку количество символов в алфавите может быть любым натуральным числом, получается, что количество таких задач, количество таких языков — это число подмножеств множества натурального ряда, и для каждого языка мы не можем подобрать распознающие алгоритм, потому что само число алгоритмов является счетным. Это было первое соображение.

Второе соображение заключалось в том, чтобы показать, что существует такая задача, понятная всем, которая не имеет алгоритмического решения. Мы рассмотрели такую задачу: существует ли алгоритм или программа, которые могут проанализировать другой алгоритм или программу и сказать, зациклится она или остановится (проблема останова). Мы рассуждали так: если гипотетически такой алгоритм существует, то мы можем на его основе сделать новый алгоритм, который сделан так, что если гипотетический алгоритм останова сигнализирует о том, что алгоритм остановится, построенный алгоритм, наоборот, не будет останавливаться — если применить этот алгоритм к самому себе, то полу-

чается противоречие, которое говорит о том, что такой программы не существует.

Теперь свяжем проблему останова с математическими задачами и покажем, что если бы у неё было алгоритмическое решение, удалось бы решить некоторые мировые проблемы, долго не поддававшиеся решению. Одна из таких задач - Большая теорема Ферма (вы знаете, что она уже решена). В чем она состоит? Рассмотрим уравнение в целых числах:

$$x^n + y^n = z^n, (x, y, z, n \in \mathbb{N})$$

Известно, что при $n = 2$ решение имеется (в качестве упражнения предлагается найти все такие решения, в том числе и в общем виде), а вот при $n \geq 3$ это было неочевидно и долгие годы проводились исследования: сначала для трёх выяснили, что решений нет, затем расширили это множество и, наконец, Уайлсом было доказано, что решений нет вообще.

Предположим теперь, что у задачи (проблемы) останова есть алгоритм. Как тогда можно было бы решать эту задачу? Мы бы написали программу, которая перебирает все четвёрки x, y, z, n и останавливается, если решение найдено, и не останавливается в ином случае. Значит, тогда мы берем нашу теоретически существующую программу, которая проверяет, остановится или нет, вводим в неё этот «переборный» алгоритм и спрашиваем, остановится он или нет. Если остановится — решение есть, не остановится — нет.

Понятно, что поскольку так легко не получилось справиться с этой проблемой, то, наверное, и задача останова не имеет решений. Кстати, интересно, сумели бы вы сами написать этот алгоритм «перебора» всех вариантов, которые мы бы предложили этому анализатору? Первая мысль, которая может у вас возникнуть: сделать четверной цикл (верхний цикл идет по n, второй — по x, следующий — по y и последний — по z). Если начать с $n = 3$, то для него придется перебирать бесконечное количество вариантов значений переменных — так цикл никогда до четырёх не дойдет, значит, это неправильная идея.

Как же нужно было бы сделать? Можно было бы реализовать идею наподобие идеи с гостиницей с бесконечным числом номеров и бесконечным числом делегаций из бесконечного числа членов, которые нужно в неё поселить (см. «Рассказы о множествах» Н.Я.Виленкина), иными словами, как пронумеровать положительные рациональные числа:

	1	2	3	4	...
1	№1	№2	№4		...
2	№3	№5			...
3	№6				...
4					...
...	...	...	...	...	...

Понятно, что для чисел, которые находятся на диагонали, сумма одинакова: $(2+1=3, 2+1=3, 3+1=4, 2+2=4, 3+1=4)$, поэтому можно было бы оттолкнуться от того, что нужно взять внешний цикл по S и уже тогда проверять все эти комбинации, не предполагая, что какой-нибудь цикл никогда не выполнится. 7:22

Algorithm 1 Большая теорема Ферма

```
s := 6;    ▷ Можно начать с шести, потому что x, y, z по крайней мере единицы.
ПОВТОРЯТЬ
for n от 3 до s do
    for x от 1 до s − n do   ▷ Двигаться дальше s − n смысла уже нет, так как сумма будет больше, чем s.
        for y от 1 до s − n − x do
            for z от 1 до s − n − x − y do
                if x^n + y^n = z^n then
                    ЗАКОНЧИТЬ
                else
                    s := s + 1
                end if
            end for
        end for
    end for
end for
```

Какие еще есть интересные математические задачи, у которых нет алгоритма? Мы с вами проходили диофантовы уравнения, и выражение $x^n + y^n = z^n$ тоже является таковым, но мы нашли алгоритм для частного случая уравнения, поэтому возникает вопрос (поставлен математиком Гильбертом): можно ли найти общий алгоритм? Ввести любой многочлен с целыми коэффициентами, а программа отвечает, имеет он решение или нет. Оказалось, что такого алгоритма не существует и последний шаг в доказательстве этого факта был сделан петербургским (тогда ленинградским) математиком Ю.В.Матиясевичем.

В рассмотренном примере мы показали, что, если бы проблема останова имела алгоритмическое решение, то и решалась бы большая теорема Ферма. Но мы не доказали эквивалентности этих задач. Более того, они и не являются эквивалентными: большая теорема Ферма доказана, несмотря на то, что задача останова неразрешима. Рассмотрим другой пример, который называется *Пасьянс Конвея* и который дает пример задачи, эквивалентной задаче останова.

Пасьянс Конвея.

13:52

Джон Конвей, создатель игры «Жизнь», которая является примером клеточного автомата, придумал такую игру, которая тоже не имеет алгоритма решения. Такой пасьянс устроен следующим образом: пусть вы имеете набор рациональных чисел $r_1, r_2, \ldots, r_n (r_i = \frac{m}{n})$; есть еще какое-то число N. Игра состоит в следующем: вы берете это число и пробуете умножить на дробь r_1. Если результат получается целый, то за числом N вы помещаете результат. Получается набор N_0, N_1. Далее выполняются те же действия уже с N_1, и если умножение не может быть произведено с целым результатом, вы переходите к r_2 и так далее.

Таким образом получается некоторая последовательность, которая может закончиться, а может не закончиться, то есть она зациклится. Возникает вопрос: существует ли алгоритм, который по набору рациональных чисел $r_1, r_2, \ldots, r_n$ и числу N скажет, остановится ли этот пасьянс или нет? Так вот, сейчас мы поставим цель на установление взаимно однозначного соответствия всех программ (алгоритмов) и всех пасьянсов, и у нас получится, что определение того остановится ли пасьянс, эквивалентна задаче останова программы (алгоритма).

Пример 1. 1.

$$r_1 = \frac{3}{4}; r_2 = \frac{2}{5}; r_3 = \frac{9}{16};$$
$$N = 10;$$

Если $N_0 = 10$, то на r_1 умножить не получается. Умножаем на r_2, имеем $N_1 = 4$. Умножаем N_1 на r_1, получаем $N_2 = 3$.

$$r_1 = \frac{3}{4}; r_2 = \frac{2}{5}; r_3 = \frac{9}{16};$$
$$N_0 = 10; N_1 = 4; N_2 = 3;$$

Далее при умножении чисел на тройку целого результата не получается, и процесс заканчивается.

2.

$$r_1 = \frac{3}{5}; r_2 = \frac{5}{3};$$
$$N = 3;$$

Если $N_0 = 3$, то на r_1 умножить не получается. Умножаем на r_2, имеем $N_1 = 5$. Умножаем N_1 на r_1, получаем $N_2 = 3$ и так далее:

$$r_1 = \frac{3}{5}; r_2 = \frac{5}{3};$$
$$N_0 = 3; N_1 = 5; N_2 = 3; \ldots$$

Это пример бесконечного пасьянса.

Итак, мы разобрались с тем, как устроен этот пасьянс, теперь нам нужно взять и сопоставить его какому-то определению алгоритма. Можно было бы попытаться использовать машину Тьюринга или алгоритм Маркова (попробуйте разобраться самостоятельно).

Приведем соображения математика Шеня на основе лекции, которая вошла в книгу «М. Вялый, В. Подольский, А. Рубцов, Д. Шварц, А. Шень. Лекции по дискретной математике», В ней показано, как сопоставить операции с пасьянсом операторам алгоритмического языка. В этой лекции сначала описывается искусственный язык Fractran. Говорится, что для самого простого языка нам достаточно выделить всего два оператора. У каждого из них есть или может быть метка, например, $q : x++$. Это один из операторов: к переменной прибавляется единица, далее идет следующий оператор с меткой q' и так далее.

Вы можете подумать, что оператор прибавления единицы малозначим для выражения всех функций, но не стоит забывать, что вся дискретная математика работает с целыми числами: например, рациональные числа — это пара целых чисел, поэтому с ними тоже можно работать, как с целыми. Вы скажете: а как сложить два числа? И вот оказывается, если использовать еще один оператор, то это можно сделать. Вторая команда такая:
$q : x-- \; if \; exeption \; go \; to \; q''$, то есть если при вычитании единицы из x не получили натуральное число, то управление передается на команду с меткой q'', а в ином случае следующей по порядку команде.
И вот оказывается, что из этих двух операторов можно построить все стандартные алгоритмические конструкции: ветвления, циклы, массивы и так далее.

Теперь возникает вопрос: как эти метки сделать уникальными? Для этого можно использовать, например, простые числа, а ещё лучше некоторое бесконечное подмножество простых чисел. Тогда каждому состоянию программы можно сопоставить уникальное натуральное число N. Оно строится так:
$N = q \cdot {p_1}^{x_1} \cdot {p_2}^{x_2} \cdot \ldots \cdot {p_n}^{x_n}$
Это значит, что в данный момент выполняем команду q, а показатели степеней дают значения всех переменных.

Обратите внимание, что если мы будем использовать в качестве q и p_i различные простые числа, то мы получим однозначное соответствие между состояниями нашей программы и натуральными числами. q и p_i должны находиться в разных подмножествах простых чисел. Например, мы можем договориться, что условно каждое четное по счету простое число используется для меток оператора. Вот такой прием дает взаимно однозначное соответствие между натуральными числами и состояниями программы.

Теперь давайте предположим, что у нас есть программа, и мы по ней хотим сделать пасьянс. Как мы будем строить набор рациональных чисел? Начать нужно с первой команды, и в качестве начального числа пасьянса будет использоваться
$N_0 = q_0 \cdot {p_1}^{{x_1}^0} \cdot \ldots \cdot {p_n}^{{x_n}^0}$, где ${x_i}^0$ — начальное значение i-ой переменной.

Построим действие пасьянса, соответствующее команде $q : x_1 + +$. Какую дробь мы ей сопоставим? Мы должны взять наше текущее состояние N для последующего умножения, и чтобы получить r_1, разделить N на q, потому что мы хотим перейти от этого состояния к другому, и умножить на q' — это означает, что если в числе N_0 у нас был множитель $q = q_0$, то следующим будет состояние $\frac{q'}{q}$. Далее, если у нас есть команда $q : x_1 + +$, то нужно увеличить значение переменной x_1 на единицу , для этого нужно умножить нашу дробь на p_1: $\frac{q' \cdot p_1}{q}$. У нас получилась первая дробь пасьянса для случая команды прибавления единицы.

Что будет, если какая-то из следующих команд имеет вид $q : x - - \; if \; exeption \; q''$? Для этого в пасьянс добавляются две дроби. Первая соответствует случаю, когда вычитание единицы возможно, вторая соответствует исключению. Строим первую дробь, для этого текущее состояние мы разделим на q и умножим на q'. Если поделить можно, то мы должны уменьшить x_1, то есть поделить на p_1: $\frac{q'}{q \cdot p_1}$. Построим вторую дробь, соответствующую исключению, когда вычитание единицы невозможно. Тогда мы переходим на команду с меткой q''. Соответствующая дробь $\frac{q''}{q}$. Вот из таких блоков мы и строим пасьянс, и получается, что когда программа работает, у нас состояния N меняются, соответсвенно, $N_0, N_1, \ldots$, пока не кончится набор рациональных чисел, и тогда вопрос остановки

программы эквивалентен остановке пасьянса. Если пасьянс останавливается, то останавливается и программа.

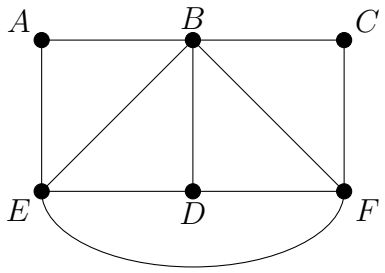
Таким образом, если бы мы нашли алгоритм, который может проанализировать этот пасьянс и дать ответ, остановится ли он, то и для любой программы мы смогли бы дать такой ответ.

Глава V. Прикладная теория алгоритмов.

31:22

§1. Алгоритмы построения минимальных остовных деревьев.

Вспомним, что такое *дерево*, и заодно сформулируем несколько новых понятий. Пусть у нас есть граф такого вида:



Сейчас мы будем рассматривать *неориентированные* графы, также будем интересоваться *связными* графами — графами, между двумя вершинами которого есть путь по ребрам.

Далее будем рассматривать *подграфы* исходного графа, то есть брать какие-то вершины и какие-то соединяющие их ребра. В частности, можно выбирать те ребра, которые образуют *дерево*. Напомним, что дерево — это связный граф, не содержащий циклов.

Определение. *Остовным деревом* графа называется его подграф, который является деревом и содержит все вершины этого графа.

Теперь мы зададим на каждом ребре его вес – числовое значение – и будем называть такой граф *взвешенным*. Поставим такой вопрос: как выбрать из всех остовных деревьев дерево минимального веса — эту задачу мы сегодня и будем обсуждать.

Для того, чтобы доказывать корректность приводимых далее алгоритмов, надо вспомнить некоторые свойства деревьев:

1. Если у нас есть одна вершина, то это уже дерево. В этом дереве одна вершина и ноль ребер. Если добавить еще одну вершину

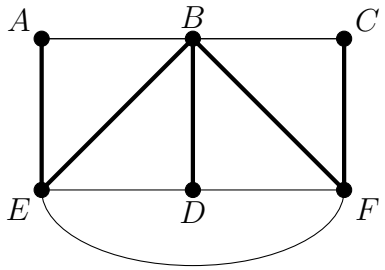
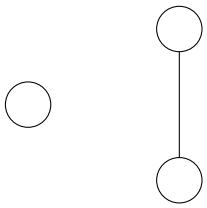


Рис. 1: Пример остовного дерева.

и соединить их ребром, получится дерево из одного ребра и двух вершин.



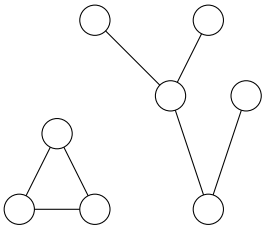
Получается, что когда мы строим дерево, мы всегда действуем так, что мы раз за разом добавляем вершину и ребро, которое связывает новую вершину с уже построенной частью. Таким образом, у дерева всегда будет выполняться соотношение: количество вершин на одну больше количества ребер.

$$B = P + 1$$

Это свойство является *характеристическим*, то есть если у вас есть связный неориентированный граф и вершин у него на единицу больше, чем ребер, то этот граф — обязательно дерево, и таким свойством можно проверять связный граф на то, является ли он деревом. Например, достаточно соединить две вершины ребром, как это свойство нарушится: ребро вы добавили, а вершину — нет, количество вершин и ребер сравнялось, и граф уже не является деревом.

Обратите внимание, если вы забудете сказать, что граф связный, то может произойти следующий фокус:

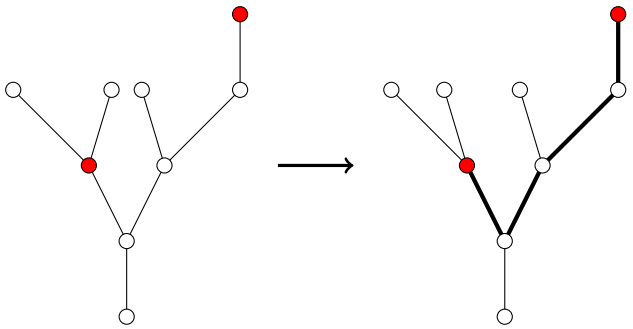
К графу с единственным циклом добавили дерево. Получился граф, состоящий из двух компонент связности, и поскольку мы добавили три вершины и три ребра, соотношение не изменилось, поэтому



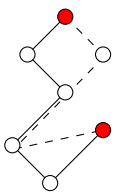
если вы скажете, что если число вершин графа на единицу больше числа ребер, то это дерево, то можете ошибиться. Но если сказать, что *связный* граф с таким свойством — дерево, то это утверждение будет верным.

2. Еще одно свойство дерева: между двумя его вершинами существует единственный путь.

   То, что путь существует, естественно, потому что граф связный. Но почему он единственный, может быть не так очевидно. Например, у нас есть две вершины (выделены красным). И мы можем построить путь между ними, видно, что он единственный:



   Как доказать, что не существует ситуации, когда есть два разных пути? Предположим, что какие-то вершины мы соединили двумя разными путями (причем пути могут частично совпадать):



   В любом случае видно, что если у нас есть совпадение каких-то вершин и от одной вершины до другой можно пройти двумя способами, мы из этих двух «кусочков» можем сделать цикл — получится, что

если есть два несовпадающих пути, то обязательно есть цикл. А это противоречит определению дерева.

Теперь мы можем сформулировать первый из алгоритмов и доказать его корректность. Вы, наверное, слышали выражение «жадный» алгоритм: его идея в том, что вы принимаете решение локально и оказывается, что глобально оно тоже дает оптимальный результат. Для построения минимального остовного дерева самый простой способ — выбирать всегда самое маленькое ребро (ребро с минимальным весом). Тогда понятно, что не возникнет вопроса, почему оно минимально. Другое дело, что в процессе выбора ребер могут появиться циклы, и это может подпортить такой способ построения минимального остовного дерева.

Алгоритм Краскала.

Алгоритм Краскала устроен следующим образом: 40:53

Algorithm 2 Алгорим Краскала.

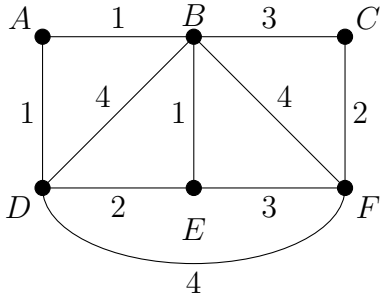
```
упорядочим ребра по возрастанию весов
for all непросмотренным рёбрам do
    выбрать ребро с минимальным весом
    if при добавлении выбранного ребра не образуется цикл then
        добавить ребро и пометить как просмотренное
    end if
end for
```

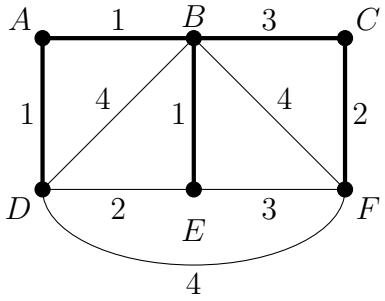
В данном алгоритме мы каждый раз выбираем самое маленькое ребро, и суммарно вес у нас получится также минимальным. Из-за того, что мы все-таки выбираем не все ребра, теоретически может получиться, что это не самый лучший вариант.

Пример 2. Вернемся к уже знакомому графу и обозначим на нем веса:

Начинаем первый шаг: берем, допустим, ребро AB, следующими — AD и BE. Потом можно выбрать DE, но это ребро образует цикл, поэтому выбираем CF. Обратите внимание, что если раньше у нас получалось дерево, то сейчас, на промежуточном этапе, это выглядит как «лес», то есть несколько деревьев.

На следующем шаге мы выберем либо BC, либо EF (будем следовать алфавитному порядку имен ребер и выберем первое ребро). Дальше уже можно проверять и остальные ребра, которые мы пропустили, но они все будут образовывать цикл.





Чем плох этот алгоритм? Первое – непонятно, как определять наличие цикла: может оказаться, что сама по себе процедура определения цикла более сложная, чем весь остальной алгоритм, и этот алгоритм перестает быть эффективным. И не сведется ли задача поиска циклов к переборной трудоемкой задаче?

Докажем сначала, что алгоритм Краскала работает корректно, а потом попробуем его уточнить так, чтобы он остался эффективным. 47:03

Теорема. *Алгоритм Краскала строит минимальное остовное дерево для любого связного взвешенного неориентированного графа.*

Доказательство. (от противного) Пусть у нас существует другое дерево, которое имеет меньший вес.

T — дерево, построенное по алгоритму Краскала;

T' — предположительно дерево меньшего веса;

$p(T) > p(T')$ — веса деревьев.

Сравним ребра. Изначально у нас они были упорядочены по возрастанию, мы идем по ним и сравниваем: допустим, первое ребро вошло и в дерево Краскала, и во второе. Смотрим следующее — оно тоже попадает в оба дерева. Рано или поздно произойдет следующее: мы найдем первое ребро, которое не входит в множество ребер T', но является очередным выбранным ребром в дереве Краскала T.

Пусть e' — первое (в порядке возрастания) ребро T, которое не вошло в T'.

Рассмотрим теперь такой граф: мы добавим к дереву T' ребро e', т.е. $T' \cup e'$. Поскольку дерево у нас было остовным, все его вершины заняты (в том смысле, что у каждой вершины есть ребро, инцидентное ей), и если мы добавим еще одно ребро, то число ребер увеличится, а число вершин — нет. Это означает, что характеристическое соотношение дерева нарушится, ведь у нас будет одинаковое количество ребер и вершин. Значит, в графе $T' \cup e'$ будет цикл, и он будет обязательно содержать добавленное ребро e', потому что без него цикла не было.

В этом цикле не могут быть только ребра, которые были просмотрены до e', потому что тогда получится, что в дереве Краскала образовался цикл. Также в нем обязательно будет ребро e большего веса, чем e', так как все ребра меньшего веса нами просмотрены.

Теперь давайте сделаем следующее:

$(T' \cup e') \setminus e = T''$ — у нас получится новое дерево T'', у которого вес будет меньше: $p(T'') < p(T')$.

Полученное противоречие доказывает минимальность дерева Краскала. Заметим, что если в графе есть ребра одинакового веса, то может быть несколько минимальных остовных деревьев, но все они будут иметь одинаковый вес (в алгоритме Краскала мы строили одно дерево, так как использовали дополнительно упорядочение ребер одинакового веса по алфавиту). □

Замечание (от редактора). Преподаватель в доказательстве поменял деревья T и T', чтобы исправить ошибку в рассуждениях.

Теперь мы рассмотрим модифицированный алгоритм, который автоматически проверяет наличие цикла. Идея состоит в следующем: мы раскрасим все вершины в разные цвета, а потом по мере соединения соединяемые ребрами вершины они окрашиваются в один цвет. И тогда получится, что если на очередном шаге вершины добавляемого ребра окрашены одним цветом, то дерево достраивается (при этом все вершины с цветами одного конца ребра перекрашиваются в цвет другого конца, если же оба конца ребра окрашены в один цвет, то добавлять ребро нельзя.

Алгоритм Краскала с раскраской вершин

55:25

Algorithm 3 Инициализация цветов вершин для алгоритма Краскала с раскраской вершин

```
k := 1                                              ▷ k — номер цвета
for all v ∈ V do    ▷ v — некоторая вершина, V — множество вершин
    color(v) := k
    k := k + 1
end for
```

Таким образом, изначально каждая вершина имеет свой цвет.

Algorithm 4 Алгоритм Краскала с раскраской вершин.

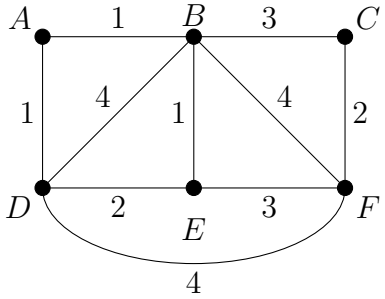
```
упорядочим ребра по возрастанию весов
for all непросмотренным рёбрам do
    выбрать минимальное ребро (u;v)
    if color(u) ≠ color(v) then
        добавить ребро
        for all w ∈ V do
            if color(w) = color(u) then
                color(w) := color(v)
            end if
        end for
    end if
    пометить ребро как просмотренное
end for
```

Замечание (от редактора). Преподаватель изменил содержание алгоритма на таймкоде справа. Предыдущий алгоритм также является правильным, но усложненным. 1:05:38

Теперь давайте посмотрим на примере, как этот алгоритм работает и как нужно оформлять задание с ним в ИДЗ. 1:03:36

Пример. В качестве примера будем рассматривать уже знакомый граф.

Для начала изображаем протокол раскраски вершин и, учитывая запись ребер в алфавитном порядке, заполняем:



№ ребра	A	B	C	D	E	F
AB	1	2	3	4	5	6
AD	2	2	3	4	5	6
BE	4	4	3	4	5	6
CF	5	5	3	5	5	6
DE	5	5	6	5	5	6
BC	5	5	6	5	5	6
EF	6	6	6	6	6	6
DF	6	6	6	6	6	6

Раскрасим сначала все вершины в разные цвета — это инициализация алгоритма — а теперь идем по основному циклу: выбираем минимальное ребро (в таблице мы их уже упорядочили и просто идем по очереди).

Итак, мы выбрали ребро AB. A и B имеют разные цвета, значит, мы добавляем это ребро в остовное дерево. И циклом по всем вершинам мы смотрим: есть ли вершины, у которых цвет совпадает с вершиной A (в нашем случае она одна, поэтому она окрасится в цвет 2 - цвет вершины B).

На следующем шаге рассматривается ребро AD: обе вершины окрашены в разные цвета. Мы договорились, что перекрашиваем левую вершину (и вершины с тем же цветом) в цвет правой.

Смотрим на BE: цвет B — 4, а E — 5. Теперь все четверки перекрашиваем в пятерки. CF: цвет C — 3, а F — 6. Все тройки перекрашиваем в шестерки. DE: обе пятерки, значит, не включаем это ребро и идем дальше. BC: все пятерки перекрашиваем в шестерки.

Дальше уже понятно, что вершины EF и DF имеют одинаковые цвета, в дальнейшем уже ничего не поменяется и ребра эти не включены.

Замечание. На самом деле, можно было и не проверять ребра до самого конца, а посчитать, сколько ребер включено: если у нас было n вершин, то после того, как $n-1$ ребро включено, дальше уже делать ничего не надо.