План лекции

- Поиск остовных деревьев в графе. Алгоритмы Прима и Краскала.Алгоритм Дейкстры и его связь с жадными алгоритмами.
- Алгоритм Флойда-Уоршалла и его связь с динамическим программированием.
- Потоки на графах. Максимальный поток.

Остовные деревья

Остовное дерево: ещё немного терминов

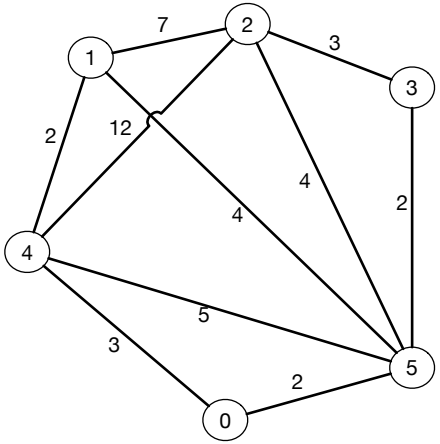
- С точки зрения теории графов **дерево** есть ациклический связный граф.
- Множество деревьев называется **лесом (forest)** или **бором**.
- **Остовное дерево** связного графа — подграф, который содержит все вершины графа и представляет собой полное дерево.
- **Остовный лес** графа — лес, содержащий все вершины графа.

Минимальное остовное дерево

- Построение остовных деревьев — одна из основных задач в компьютерных сетях.
- Решение задачи — как спланировать маршрут от одного узла сети до других.
- Для некоторого типа узлов в передаче сообщений недопустимо иметь несколько возможных маршрутов. Например, если компьютер соединён с маршрутизатором по Wi-Fi и Ethernet одновременно, то в некоторых операционных системах сообщения от компьютера до маршрутизатора не будут доходить из-за наличия цикла.
- Построение остовного дерева — избавление от циклов в графе.

Остовные деревья

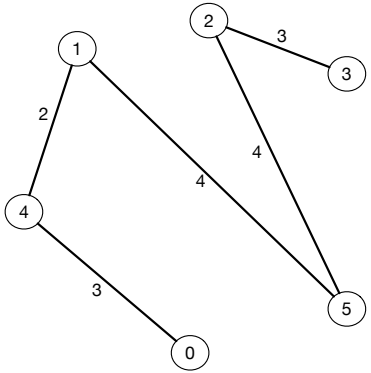
Каждый из узлов имеет информацию о связях с соседями (рёбрах). Каждое ребро имеет вес.



Остовное дерево

- Множество достижимых узлов из некоторого *корневого* узла P_r должно совпадать с полным множеством.
- Для каждого узла должен быть ровно один маршрут до любого из достижимых узлов.

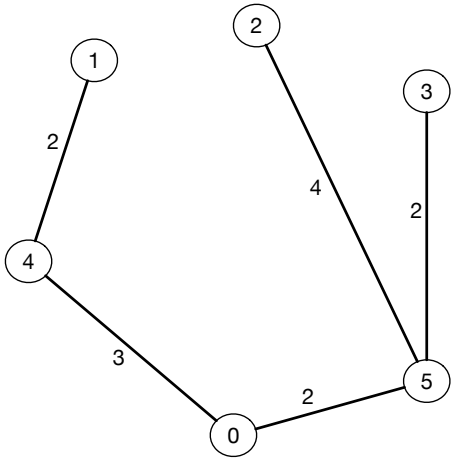
Это — остовное дерево для корневого узла P_r.



Минимальное остовное дерево

Задача: определение кратчайшего пути из корневого узла.

Минимальное остовное дерево:



Минимальное остовное дерево

- `MST` — Minimal Spanning Tree.
- Минимальное остовное дерево взвешенного графа есть остовное дерево, вес которого (сумма его всех рёбер) не превосходит вес любого другого остовного дерева.
- Именно минимальные остовные деревья больше всего интересуют проектировщиков сетей.
- **Сечение графа** — разбиение множества вершин графа на два непересекающихся подмножества.
- **Перекрёстное ребро** — ребро, соединяющее вершину одного множества с вершиной другого множества.

Свойства MST

- **Лемма.** Если T — произвольное остовное дерево, то добавление любого ребра e между двумя вершинами u и v создаёт цикл, содержащий вершины u, v и ребро e.

Свойства MST

- **Лемма.** При любом сечении графа каждое минимальное перекрёстное ребро принадлежит некоторому MST-дереву и каждое MST-дерево содержит перекрёстное ребро.
- **Доказательство** от противного. Пусть e — минимальное перекрёстное ребро, не принадлежащее ни одному MST и пусть T — MST дерево, не содержащее e. Добавим e в T. В этом графе есть цикл, содержащий e и он содержит ребро e', с весом, не меньшим e. Если удалить e', то получится остовное дерево не большего веса, что противоречит условию минимальности T или предположению, что e не содержится в T.

Свойства MST

- **Следствие.** Каждое ребро дерева MST есть минимальное перекрёстное ребро, определяемое вершинами поддеревьев, соединённых этим ребром.

Свойства MST

- **Лемма (без доказательства).** Пусть имеется граф G и ребро e. Пусть граф G' есть граф, полученный добавлением ребра e к графу G. Результатом добавления ребра e в MST графа G и последующего удаления максимального ребра из полученного цикла будет MST графа G'.
- Эта лемма выявляет рёбра, которые не должны входить в MST.

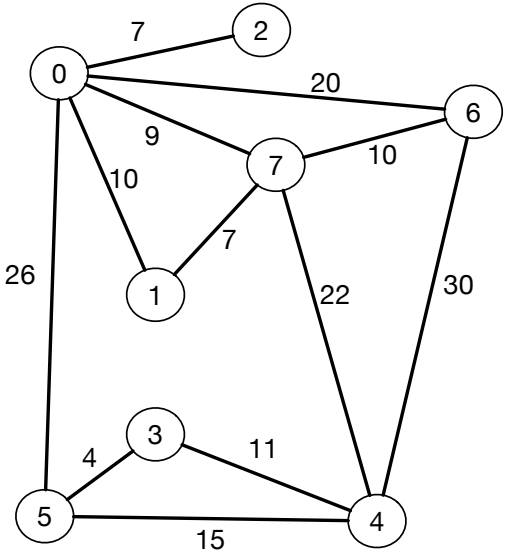
Алгоритмы поиска MST

Алгоритм Прима

1. Используется сечение графа на два подграфа — древесных вершин и недревесных вершин.
2. Выбираем произвольную вершину. Это — MST дерево, состоящее из одной древесной вершины.
3. Выбираем минимальное перекрёстное ребро между MST множеством и недревесным множеством.
4. Повторяем операцию до тех пор, пока все вершины не окажутся в дереве.

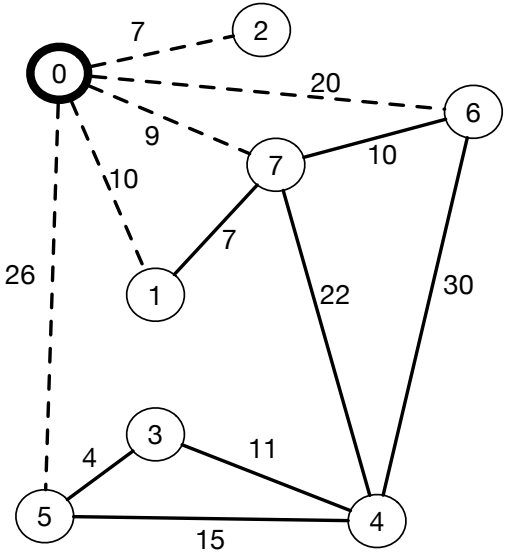
Алгоритм Прима

Исходный граф.



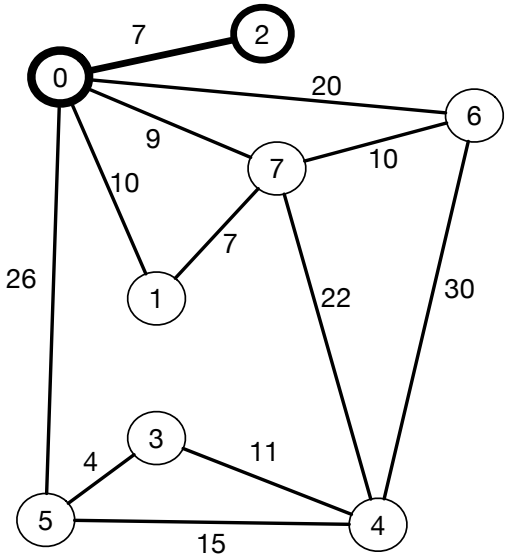
Алгоритм Прима

Вершина 0 — корневая. Переводим её в MST. Проверяем все веса из MST в не MST.



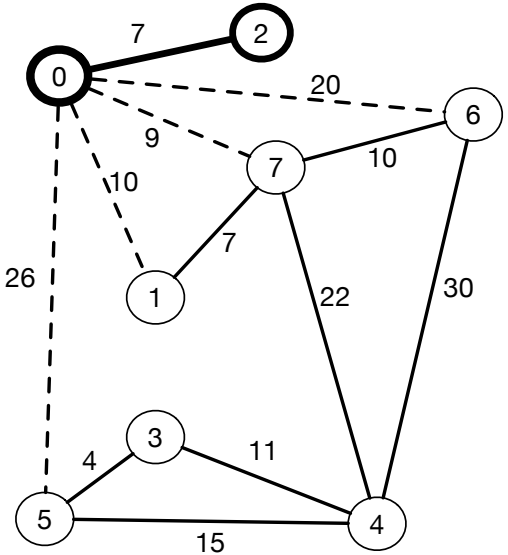
Алгоритм Прима

(0-2) самое лёгкое ребро. Переводим вершину 2 и ребро (0-2) в MST.



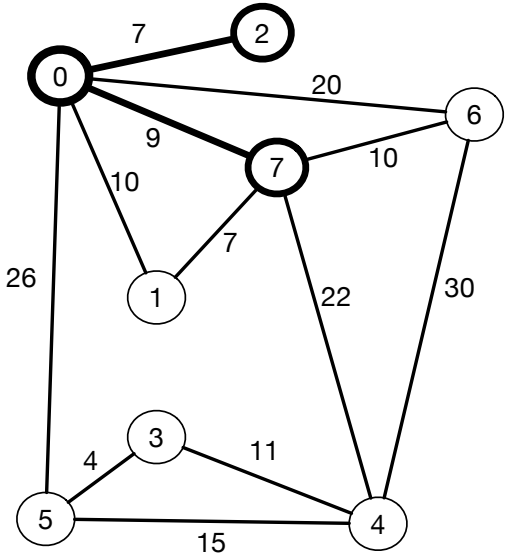
Алгоритм Прима

Отмечаем все рёбра из MST в не MST.



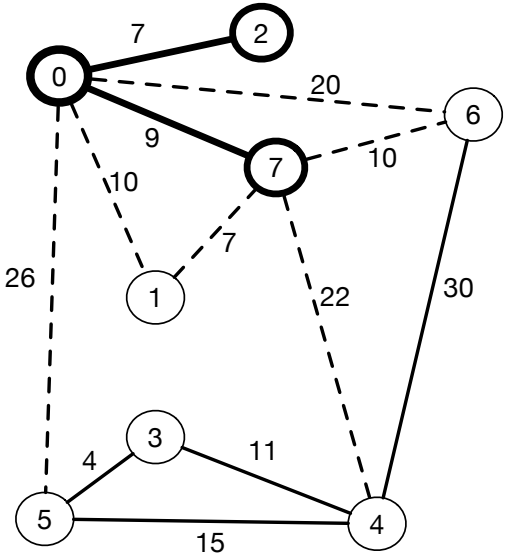
Алгоритм Прима

Переносим вершину 7 и ребро (0-7) в MST.



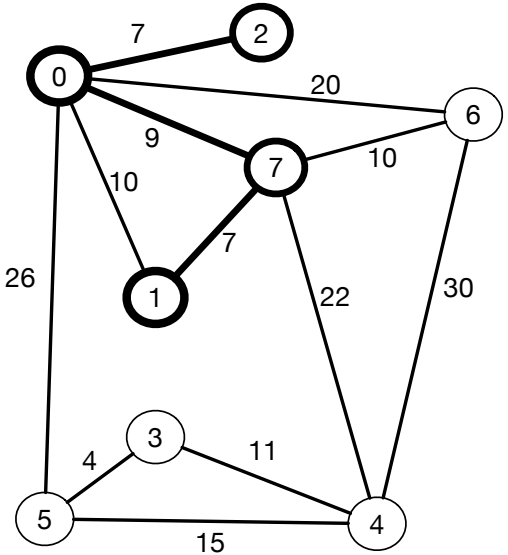
Алгоритм Прима

Отмечаем все рёбра из MST в не MST.



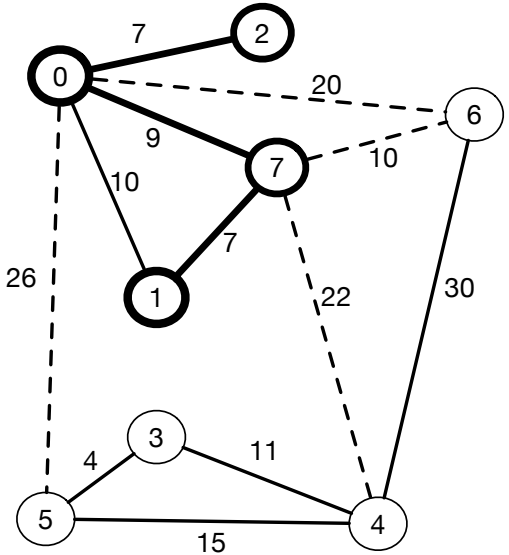
Алгоритм Прима

Переносим вершину 1 и ребро (1-7) в MST.



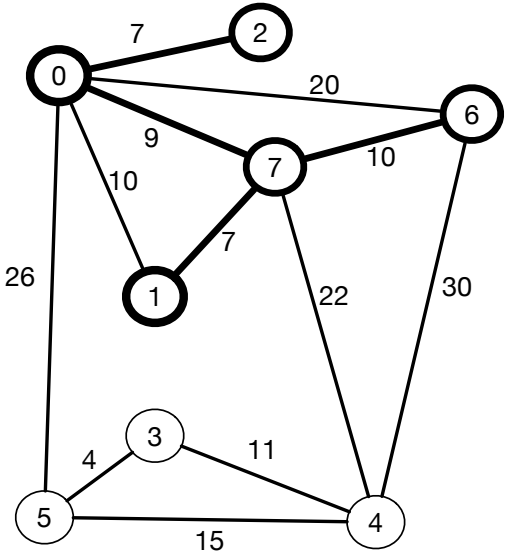
Алгоритм Прима

Отмечаем все рёбра из MST в не MST.



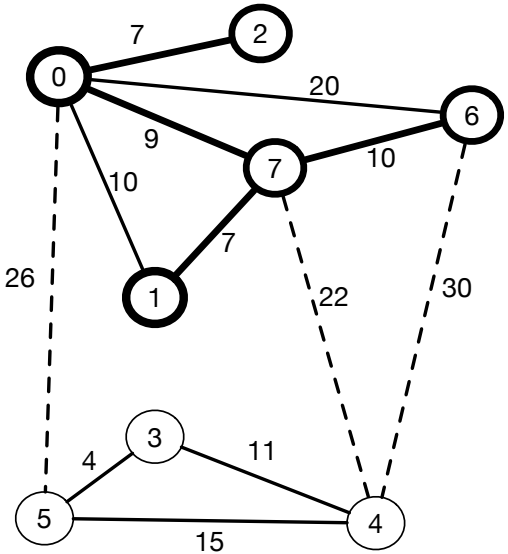
Алгоритм Прима

Переносим вершину 6 и ребро (7-6) в MST.



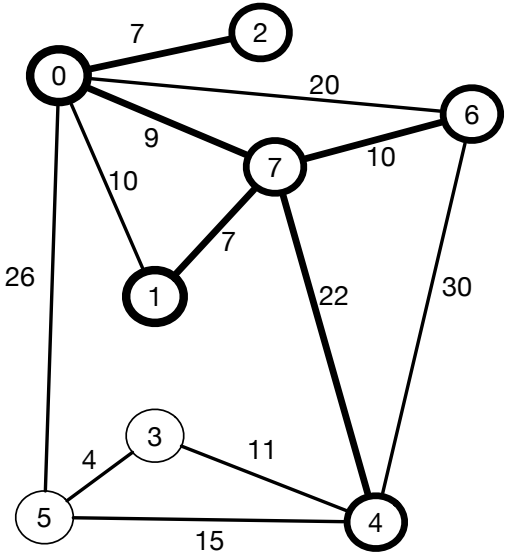
Алгоритм Прима

Отмечаем все рёбра из MST в не MST.



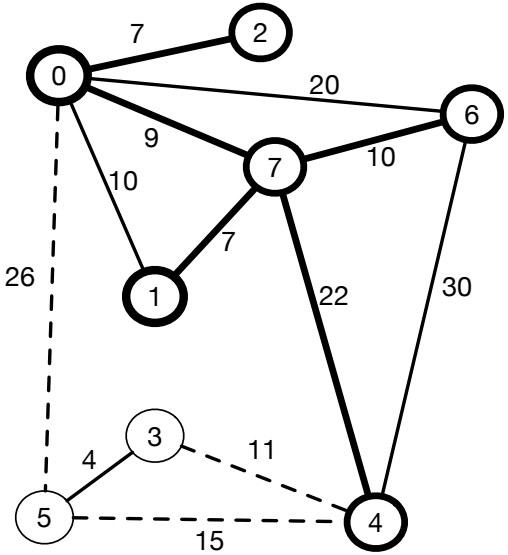
Алгоритм Прима

Переносим вершину 4 и ребро (7-4) в MST.



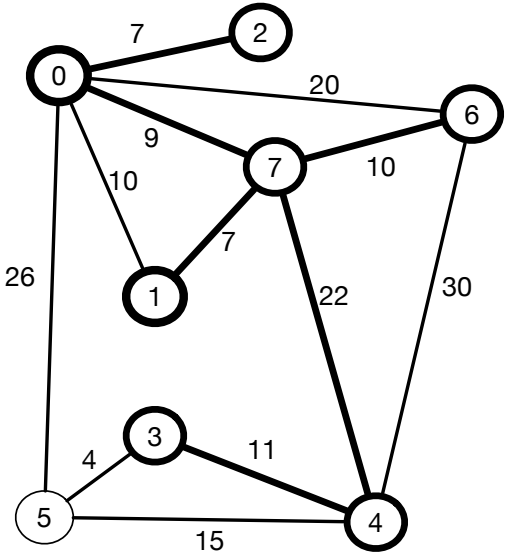
Алгоритм Прима

Отмечаем все рёбра из MST в не MST.



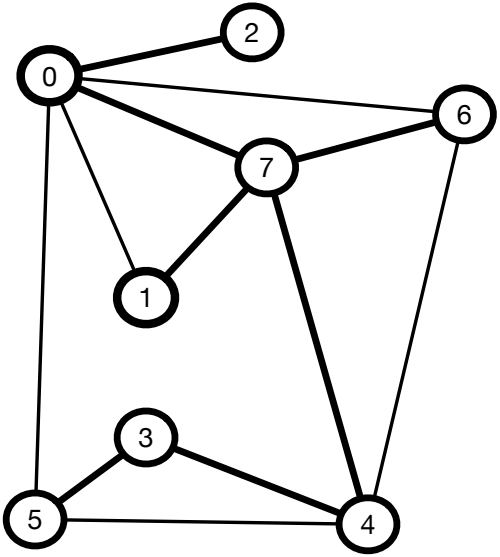
Алгоритм Прима

Переносим вершину 3 и ребро (3-4) в MST.



Алгоритм Прима

Все вершины в MST.



Алгоритм Прима

- В данном виде алгоритм не очень эффективен.
- На каждом шаге мы забываем про те рёбра, который уже проверяли.
- Введём понятие **накопителя**.
- Накопитель содержит множество рёбер-кандидатов.
- Каждый раз в `MST` включается самое лёгкое ребро.

Алгоритм Прима

Более эффективная реализация алгоритма Прима

1. Выбираем произвольную вершину. Это — MST дерево, состоящее из одной вершины. Делаем вершину текущей.
2. Помещаем в накопитель все рёбра, которые ведут из этой вершины в не MST узлы. Если в какой-либо из узлов уже ведёт ребро с большей длиной, заменяем его ребром с меньшей длиной.
3. Выбираем ребро с минимальным весом из накопителя.
4. Повторяем операцию до тех пор, пока все вершины не окажутся в дереве.

Алгоритм Прима

- Алгоритм Прима — обобщение поиска на графе.
- Накопитель представляется очередью с приоритетами.
- Используется операция «извлечь минимальное».
- Используется операция «увеличить приоритет».
- Такой поиск на графе называется PFS — поиск по приоритету.
- Сложность алгоритма $O(|E| \log |V|)$.

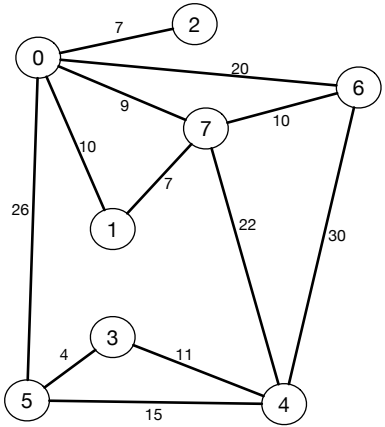
Алгоритм Краскала

Алгоритм Краскала (Kruscal).

- Один из самых старых алгоритмов на графах (1956).
- Предварительное условие: связность графа.

1. Создаётся число непересекающихся множеств по количеству вершин и каждая вершина составляет своё множество.
2. Множество `MST` вначале пусто.
3. Из всех рёбер, не принадлежащих `MST` выбирается самое короткое из всех рёбер, не образующих цикл. Вершины ребра должны принадлежать различным множествам.
4. Выбранное ребро добавляется к множеству `MST`
5. Множества, которым принадлежат вершины выбранного ребра, сливаются в единое.
6. Если размер множества `MST` стал равен $|V| - 1$, то алгоритм завершён, иначе отправляемся к пункту 3.

Алгоритм Краскала



Список рёбер упорядочен по возрастанию:

i	3	0	1	0	0	6	3	4	0	0	4
j	5	2	7	7	1	7	4	5	6	5	6
W_{ij}	4	7	7	9	10	10	11	15	22	26	30

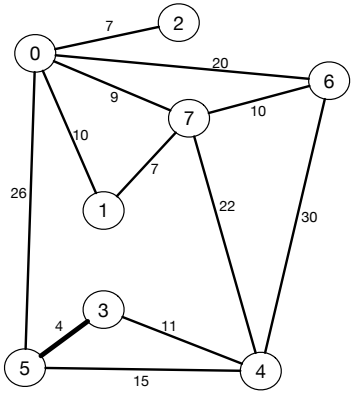
Алгоритм Краскала

Таблица принадлежности вершин множествам:

V_i	0	1	2	3	4	5	6	7
p	0	1	2	3	4	5	6	7

Алгоритм Краскала:первая итерация

Вершины 3 и 5 самого короткого ребра в разных множествах $\rightarrow$ отправляем ребро в множество MST и объединяем множества.



V_i	0	1	2	3	4	5	6	7
p	0	1	2	3	4	**3**	6	7

Алгоритм Краскала:вторая итерация

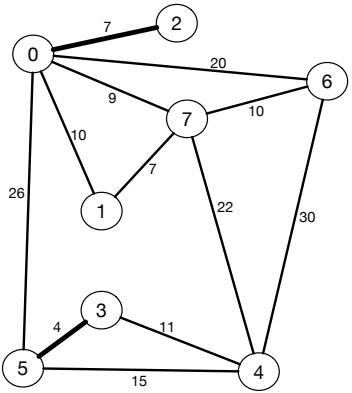
Два подходящих ребра с одинаковым весом:

i	0	1	0	0	6	3	4	0	0	4
j	2	7	7	1	7	4	5	6	5	6
W_{ij}	7	7	9	10	10	11	15	22	26	30

Лемма. При равных подходящих рёбрах можно выбирать произвольное.
Доказательство. Если добавление первого ребра не помешает добавлению второго, то, всё OK.
Если помешает (добавление второго создаст цикл), то можно удалить любое из них, общий вес дерева останется неизменным.

Алгоритм Краскала:вторая итерация

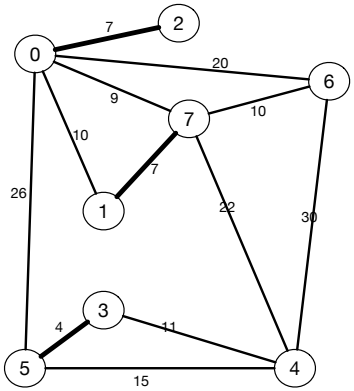
Выберем ребро (0,2) и поместим вершину 2 в множество номер 0.



V_i	0	1	2	3	4	5	6	7
p	0	1	**0**	3	4	3	6	7

Алгоритм Краскала:третья итерация

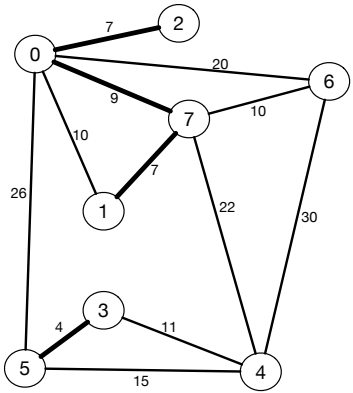
Ребро (1,7) привело к слиянию множеств 1 и 7.



V_i	0	1	2	3	4	5	6	7
p	0	1	0	3	4	3	6	**1**

Алгоритм Краскала:четвёртая итерация

Самым коротким ребром из оставшихся оказалось ребро (0,7).



Нам нужно слить два множества — одно, содержащее $\{0, 2\}$ и другое — содержащее $\{1, 7\}$.

Алгоритм Краскала: четвёртая итерация

V_i	0	1	2	3	4	5	6	7
p	0	1	0	3	4	3	6	**1**

- Пусть новое множество получит номер 0.
- Нужно ли найти в массиве p все единицы (номер второго множества) и заменить их на нули (номер того множества, куда переходят элементы первого)?
- Можно быстрее, используя *систему непересекающихся множеств* `Union-Find` или `Disjoint Set Union`, `DSU`.

Система непересекающихся множеств, DSU

Абстракция DSU реализует три операции:

- `create(n)` — создать набор множеств из n элементов.
- `find_root(x)` — найти представителя множества.
- `merge(l,r)` — сливает два множества `l` и `r`.

Система непересекающихся множеств: find_root(x)

V_i	0	1	2	3	4	5	6	7
p	0	1	0	3	4	3	6	1

- p[7]==1 — номер множества, он же и *представитель*.
- Если для слияния множеств {0,2} и {1,7} поместим в p[7] число 0, то для вершины 1 представителем останется 1, что неверно (после слияния вершина 1 должна принадлежать множеству 0).
- Так как p[7]==1, то и у седьмой, и у первой вершины представители одинаковые.
- Если номер вершины совпадает с номером представителя, то, в массив p при исполнении ничего не было записано $\rightarrow$ эта вершина есть корень дерева.

Система непересекающихся множеств

- После слияния нужно заменить всех родителей вершины на нового представителя. Это делается изящным рекурсивным алгоритмом:

```
int find_root(int r) {
    if (p[r] == r) return r; // A trivial case
    return p[r] = find_root(p[r]); A recursive case
}
```

Система непересекающихся множеств

- `merge(l,r)`. Для сохранения корректности алгоритма вполне достаточно любого из присвоений: `p[l] = r` или `p[r] = l`. Всю дальнейшую корректировку родителей в дальнейшем сделает метод `find_root`.
- Приёмы балансировки деревьев:
  - Использование ещё одного массива, хранящего длины деревьев: слияние производится к более короткому дереву.
  - Случайный выбор дерева-приёмника.

```
void merge(int l, int r) {
  l = find_root(l); r = find_root(r);
  if (rand() % 2) p[l] = r;
  else p[r] = l;
}
```

- Важно: операция слияния начинается с операции поиска, которая заменяет аргументы значениями корней их деревьев!

Алгоритм Краскала

V_i	0	1	2	3	4	5	6	7
p	0	1	0	3	4	3	6	1

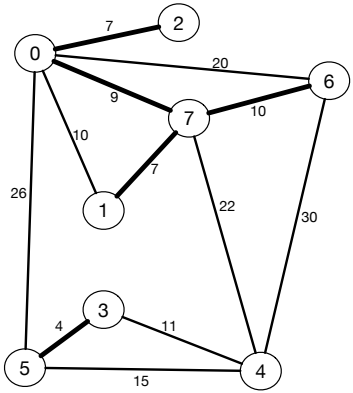
- Определяем, каким деревьям принадлежат концы ребра `(0,7)`.
- `find_root(0)` вернёт 0 как номер множества.
- `find_root(7)` сначала убедится, что в `p[7]` лежит 1 и вызовет `find_root(1)`, после чего, возможно, заменит `p[7]` на 1 и вернёт 1.

Алгоритм Краскала

- Концы ребра 0 принадлежат разным множествам $\rightarrow$ сливаем множества, вызвав `merge(0,7)`.
- Операция `merge` — заменит свои аргументы, 0 и 7, корнями деревьев, которым принадлежат 0 и 7, то есть, 0 и 1 соответственно.
- В `p[1]` помещается 0 и деревья слиты.
- Обратите внимание на то, что в `p[7]` всё ещё находится 1!.

V_i	0	1	2	3	4	5	6	7
p	0	**0**	0	3	4	3	6	1

Алгоритм Краскала

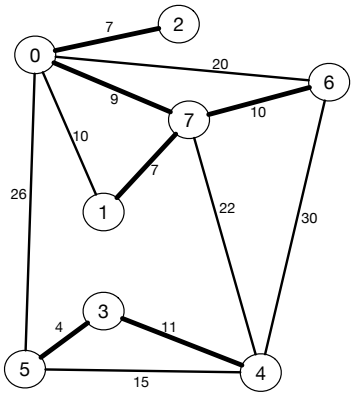


Следующее ребро — (6,7). `find_root(7)` установит `p[7]=0`. Это же значение будет присвоено и `p[6]`.

V_i	0	1	2	3	4	5	6	7
p	0	**0**	0	3	4	3	**0**	0

Алгоритм Краскала

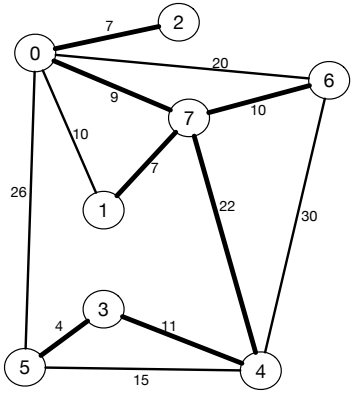
На следующем этапе ребро (3,4) окажется самым коротким.



V_i	0	1	2	3	4	5	6	7
p	0	0	0	3	3	3	0	0

Алгоритм Краскала

Концы рёбер (4,5) и (0,6) принадлежат одним множествам. Ребро (4,7) подходит. Количество рёбер в множестве MST достигло $7 = N - 1$. Конец.



V_i	0	1	2	3	4	5	6	7
p	0	0	0	3	0	3	0	0

Алгоритм Краскала: сложность

- Первая часть алгоритма — сортировка рёбер. Сложность этой операции $O(|E| \log |E|)$.
- В 1984 году Tarjan доказал, используя функцию Аккермана, что операция поиска в `DSU` имеет сложность амортизированную `O(1)`.
- Сложность всего алгоритма Краскала и есть $O(|E| \log |E|)$.
- Для достаточно разреженных графов он обычно быстрее алгоритма Прима, для заполненных — наоборот.

Алгоритм Дейкстры.

Дерево кратчайших путей — SPT

- Пусть задан граф G и вершина s. **Дерево кратчайших путей** для s — подграф, содержащий s и все вершины, достижимые из s, образующий направленное поддерево с корнем в s, где каждый путь от вершины s до вершины u является кратчайшим из всех возможных путей.

Алгоритм Дейкстры

- Строит SPT (Shortest Path Tree).
- Определяет длины кратчайших путей от заданной вершины до остальных.
- **Обязательное условие:** граф не должен содержать рёбер с отрицательным весом.

Алгоритм Дейкстры

1. В SPT заносится корневой узел (исток).
2. На каждом шаге в SPT добавляется одно ребро, которое формирует кратчайший путь из истока в не-SPT.
3. Вершины заносятся в SPT в порядке их расстояния по SPT от начальной вершины.

Алгоритм Дейкстры

Жадная стратегия.

- Пусть найдено оптимальное множество U.
- Изначально оно состоит из вершины s
- Длины кратчайших путей до вершин множества обозначим, как $d(s, v), v \in U$.
- Среди вершин, смежных с U находим вершину $u, u \notin U$ такую, что достигается минимум
$$\min_{v \in U, u \notin U} d(s, v) + w(v, u).$$
- Обновляем множество $U : U \leftarrow U \cup \{u\}$ и повторяем операцию.

Алгоритм Дейкстры

Используются переменные:

- $d[u]$ — длина кратчайшего пути из вершины s до вершины u.
- $\pi[u]$ — предшественник u в кратчайшем пути от s.
- $w(u, v)$ — вес пути из u в v (длина ребра, вес ребра, метрика пути).
- Q — приоритетная по значению d очередь узлов на обработку.
- U — множество вершин с уже известным финальным расстоянием.

Алгоритм Дейкстры

```
1: procedure DIJKSTRA(G : Graph; w : weights; s : Vertex)
2:     for all v ∈ V do
3:         d[v] ← ∞
4:         π[v] ← nil
5:     end for
6:     d[s] ← 0
7:     U ← ∅
8:     Q ← V
9:     while Q ≠ ∅ do
10:        u ← Q.extractMin()
11:        U ← U ∪ {u}
12:        for all v ∈ Adj[u], v ∉ U do
13:            Relax(u,v)
14:        end for
15:    end while
16: end procedure
```

Алгоритм Дейкстры

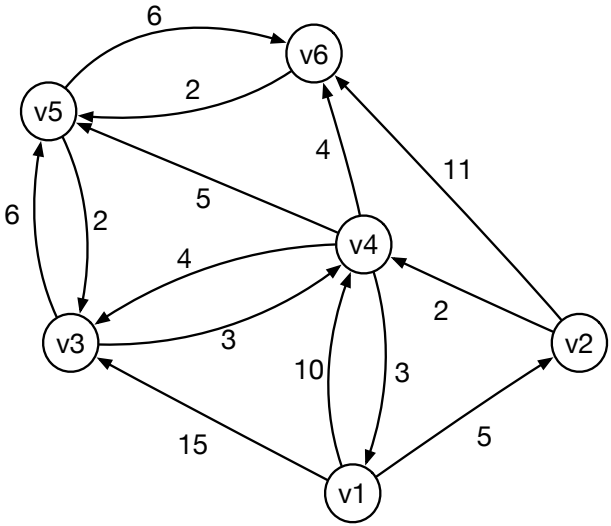
1: **procedure** $\textsc{Relax}(u, v : Vertex)$
2: **if** $d[v] > d[u] + w(u, v)$ **then**
3: $d[v] = d[u] + w(u, v)$
4: $\pi[v] \leftarrow u$
5: **end if**
6: **end procedure**

Алгоритм Дейкстры

- Операция $Relax$ — релаксация
- Два вида релаксации:
  - Релаксация ребра. Даёт ли продвижение по данному ребру новый кратчайший путь?
  - Релаксация пути. Даёт ли прохождение через данную вершину новый кратчайший путь, соединяющий две другие заданные вершины.

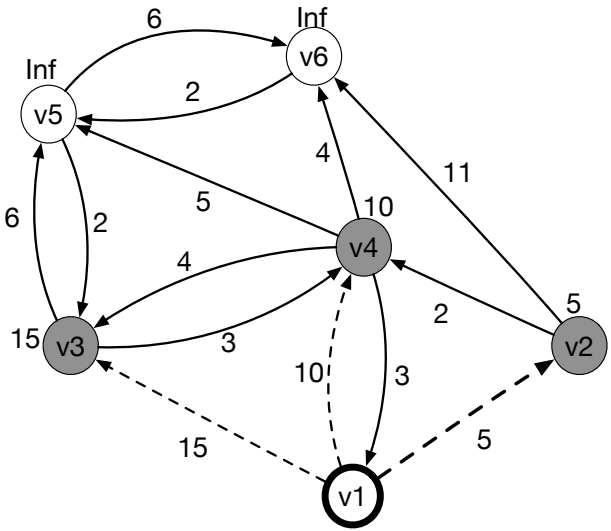
Алгоритм Дейкстры

Исходный граф



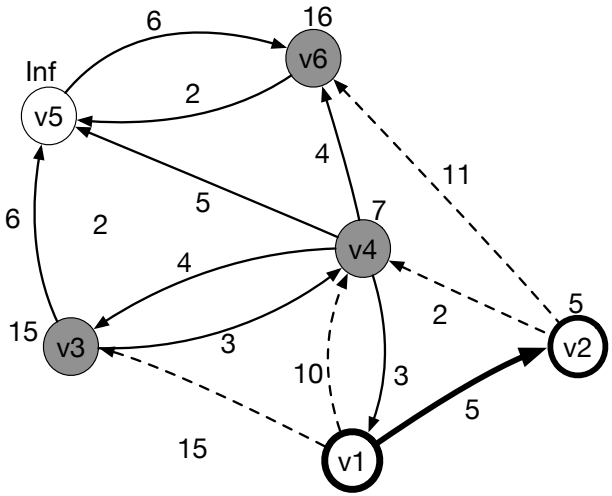
Алгоритм Дейкстры

v1 в SPT, v2, v3 и v4 — в накопителе.



Алгоритм Дейкстры

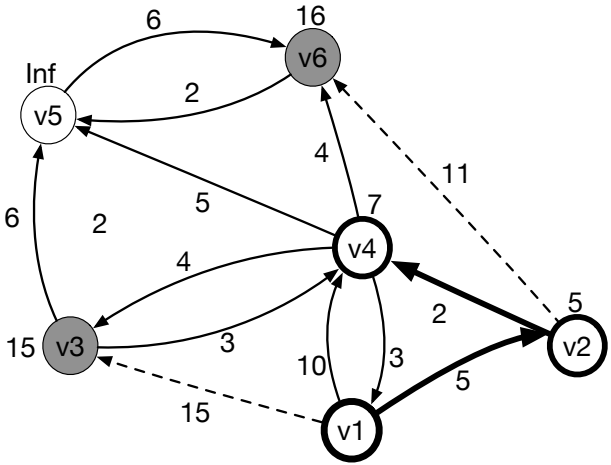
Выбран узел v2. Корректируются расстояния от него. Релаксация: $(1 \to 4)$



заменён на $(1 \to 2 \to 4)$.

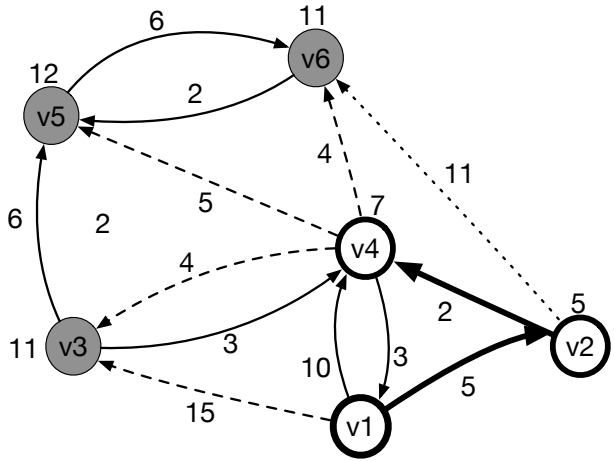
Алгоритм Дейкстры

Выбран узел v3.

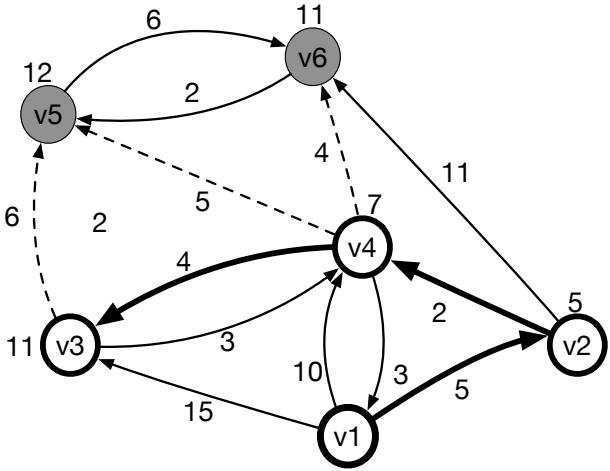


Алгоритм Дейкстры

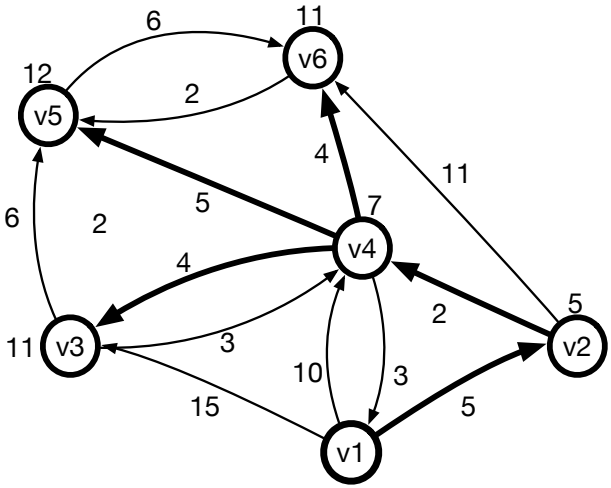
В накопитель отправляется v5. Релаксация: $(1 \to 2 \to 6)$ заменено на $(1 \to 2 \to 4 \to 6)$, $(1 \to 3)$ на $(1 \to 2 \to 4 \to 3)$



Алгоритм Дейкстры



Алгоритм Дейкстры



Алгоритм Дейкстры: сложность

- Имеется $|V| - 1$ шаг.
- На каждом шаге корректировка расстояние до соседей (просмотреть все рёбра) и выбор минимального из накопителя.
- Для насыщенных деревьев сложность алгоритма $O(V^2 \log V)$

Множественный алгоритм Дейкстры

- Если мы хотим построить таблицу минимальных расстояний от каждого до каждого, то вычисление таблиц для каждого узла в отдельности имеет сложность $O(N^2 \log N)$.
- Вычисление таблиц для всех узлов имеет сложность $N \cdot O(N^2 \log N) = O(N^3 \log N)$
- Существует более быстрый алгоритм, имеющий сложность $O(N^3)$.

Алгоритм Флойда-Уоршалла.

Алгоритм Флойда-Уоршалла

Построение таблиц маршрутизации.

- Известен с 1962 года.
- Определяет кратчайшие пути во взвешенном графе, описанном матрицей смежности.
- В матрице смежности число, находящееся в i-й строке и j-м столбце есть вес связи между ними.
- Изменим представление и будем полагать, что в матрице смежности $C_{ij} = \infty$, если узлы i и j не являются соседями.
- На входе алгоритм принимает модифицированную матрицу смежности, а на выходе эта матрица будет содержать в элементе C_{ij} вес кратчайшего пути из P_i в P_j.
- Допускается наличие путей с отрицательным весом.
- Не должно быть циклов с отрицательной длиной.

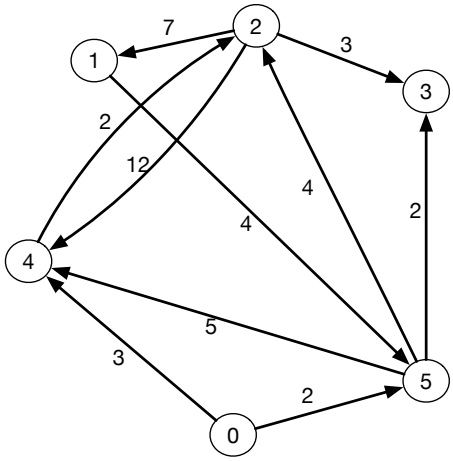
Алгоритм Флойда-Уоршалла

Сам алгоритм может быть описан в рекурсивной форме как

$$D_{ij}^{(k)} = \begin{cases} C_{ij}, & \text{если } k = 0, \\ \min\left(D_{ij}^{(k-1)}, D_{ik}^{(k-1)} + D_{kj}^{(k-1)}\right), & \text{если } k \geqslant 1 \end{cases}$$

Это — задача динамического программирования.

Этапы прохождения алгоритма для графа



Алгоритм Флойда-Уоршалла

Исходная матрица смежности:

$D^{(0)} =$

	P_0	P_1	P_2	P_3	P_4	P_5
P_0	0	∞	∞	∞	3	2
P_1	∞	0	∞	∞	∞	4
P_2	∞	7	0	3	12	∞
P_3	∞	∞	∞	0	∞	∞
P_4	∞	∞	2	∞	0	∞
P_5	∞	∞	4	2	5	0

Начальная матрица $D^{(0)}$ содержит метрики всех наилучших маршрутов единичной длины. Каждая следующая итерация алгоритма добавляет в матрицу $D^{(i+1)}$ элементы, связанные с маршрутами длины i, на единицу большей.

Алгоритм Флойда-Уоршалла

После первой итерации матрицы не изменяются.
После второй итерации получается следующее (красным цветом помечены изменившиеся элементы таблиц):

$D^{(2)} =$

	P_0	P_1	P_2	P_3	P_4	P_5
P_0	0	∞	∞	∞	3	2
P_1	∞	0	∞	∞	∞	4
P_2	∞	7	0	3	12	**11**
P_3	∞	∞	∞	0	∞	∞
P_4	∞	∞	∞	∞	0	∞
P_5	∞	∞	4	2	5	0

Алгоритм Флойда-Уоршалла

$D^{(3)} =$

	P_0	P_1	P_2	P_3	P_4	P_5
P_0	0	∞	∞	∞	3	2
P_1	∞	0	∞	∞	∞	4
P_2	∞	7	0	3	12	11
P_3	∞	∞	∞	0	∞	∞
P_4	∞	∞	∞	∞	0	∞
P_5	∞	**11**	4	2	5	0

Алгоритм Флойда-Уоршалла

Результат четвёртой и пятой итерации совпадает с результатом третьей. Шестая, последняя итерация:

$$D^{(6)} = \begin{array}{c|cccccc} & P_0 & P_1 & P_2 & P_3 & P_4 & P_5 \\ \hline P_0 & 0 & \mathbf{13} & \mathbf{6} & \mathbf{4} & 3 & 2 \\ P_1 & \infty & 0 & \mathbf{8} & \mathbf{6} & \mathbf{9} & 4 \\ P_2 & \infty & 7 & 0 & 3 & 12 & 11 \\ P_3 & \infty & \infty & \infty & 0 & \infty & \infty \\ P_4 & \infty & \infty & \infty & \infty & 0 & \infty \\ P_5 & \infty & 11 & 4 & 2 & 5 & 0 \end{array}$$

Поиск максимального потока.

Поиск максимального потока

- Предположим, что мы сидим за рулём автомобиля.
- Дорожная сеть — граф.
- Алгоритм Дейкстры определит, за какое время мы доберёмся до любого пункта назначения.
- Как определить все ли автомобили могут проехать по данному маршруту, или пропускная способность транспортной сети ограничена?
- Москва 9 мая: все хотят попасть в центр на парад, но часть дорог вообще перекрыта, а часть — имеет ограниченную ширину.
- Как узнать максимальное число автомобилей, которые могут проехать в центр за, скажем, один час?
- Требуется найти **максимальный поток** между стартом и финишем, источником и стоком.

Поиск максимального потока

- Толчок: вторая мировая война, Д. Б. Данциг, отдел статистического управления ВВС США.
- Нужна математическая модель, каким образом можно быстро сконцентрировать войска и войсковую инфраструктуру вблизи критических точек на театре военных действий.
- Более общая задача: определения пропускной способности рёбер транспортного графа поставлена им в 1951 году.
- 1955 год: Лестер Форд и Делберт Фалкерсон разработали алгоритм, решающий именно эту задачу.
- Хорошее решение данной задачи критически важно для современных транспортных графов (Москва: более 100000 узлов).

Поиск максимального потока: термины

- **Ёмкость ребра** — максимальная интенсивность потока, проходящего через ребро.
- **Насыщенное ребро** — ребро, по которому проходит максимальный поток.

Поиск максимального потока: алгоритм

Алгоритм ищет максимальный поток в сети из источника (*source*) в сток (*destination*).

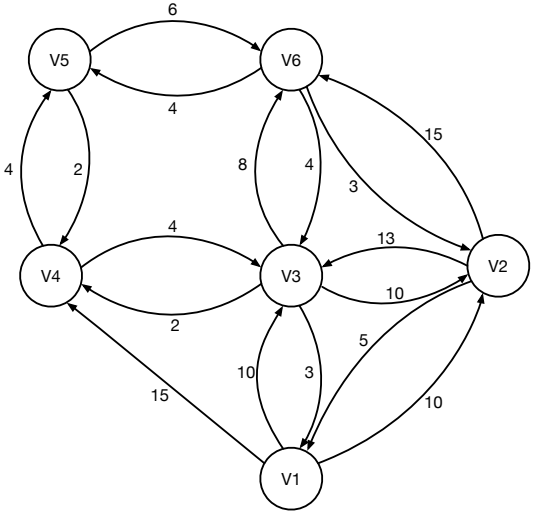
- Каждому ребру ставится в соответствие пара чисел (c, l).
- c — достигнутый до сих пор поток по ребру, вначале он равен нулю, затем это число будет только увеличиваться, пока не достигнет l, ёмкости ребра.
- Если по ребру (u, v) мы пустили прямой поток, пустим такой же в обратном направлении (v, u), добавив, если надо, отсутствующее ребро.
- Алгоритм продолжается, когда на хотя бы одном маршруте из s все рёбра — ненасыщенные, то есть на всех рёбрах $c < l$.

Поиск максимального потока

1. Ищем любой маршрут, содержащий только ненасыщенные рёбра из s в d. Если такого нет, то алгоритм закончен, искомый поток есть сумма потоков всех рёбер, приходящих в d.
2. Мы нашли **дополняющий маршрут**. Определяем значение максимального потока m, который мы можем пропустить по данному маршруту. Он определяется как минимальная из всех возможных разностей ёмкости l и существующего потока c по всем рёбрам маршрута.
3. К каждому из c на маршруте прибавляем m. Хотя бы одно ребро станет насыщенным.
4. Возвращаемся к (1).

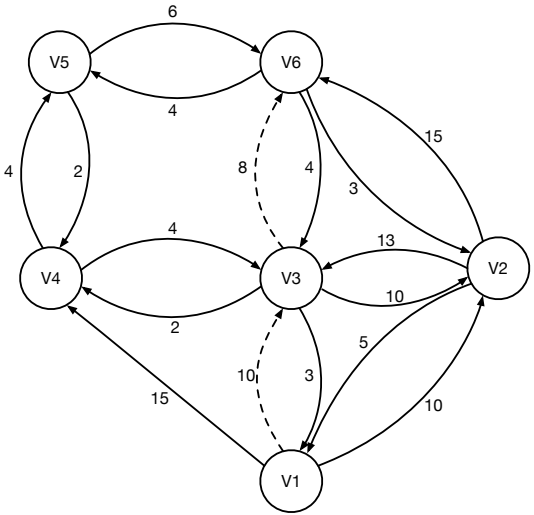
Поиск максимального потока: подопытный граф.

Ищем максимальный поток из V_1 в V_6.



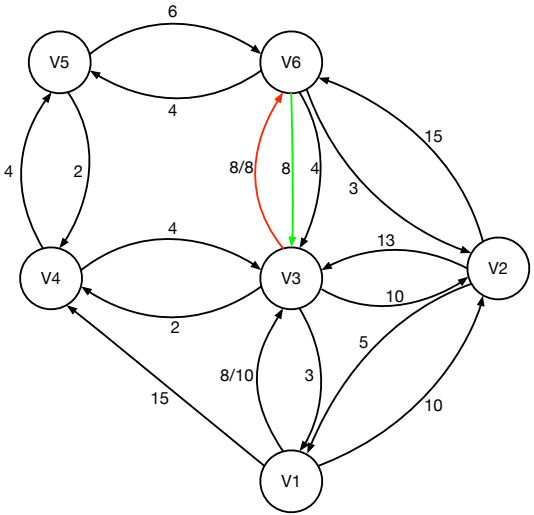
Поиск максимального потока

- Найдём произвольный маршрут из s в d. Пусть $v1 \to v3 \to v6$.
- Наименьшая разница между l и c равна 8, пропускаем поток с интенсивностью 8 по этому маршруту.



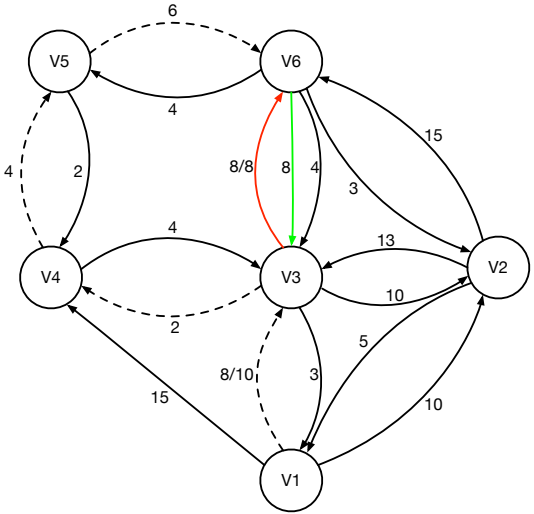
Поиск максимального потока

- Ребро $V_3 \to V_6$ стало насыщенным.
- Добавим обратное ребро $V_6 \to V_3$.



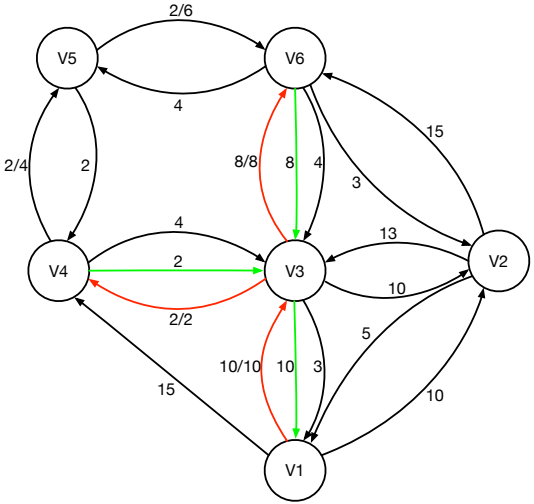
Поиск максимального потока

- Находим новый путь из V_1 в V_6.



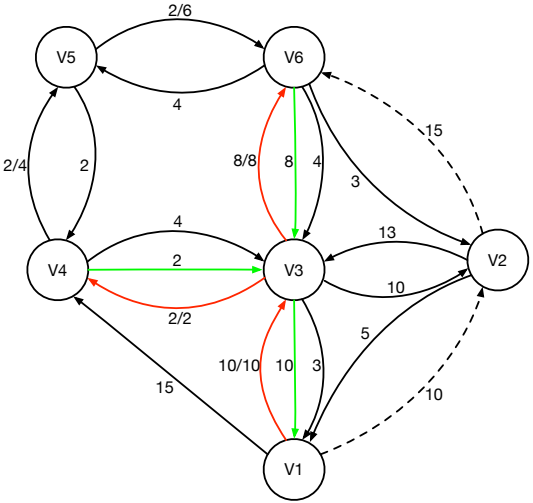
Поиск максимального потока

- Он сделал насыщенными рёбра $V_1 \to V_3$ и $V_3 \to V_4$.
- Добавим обратные рёбра.



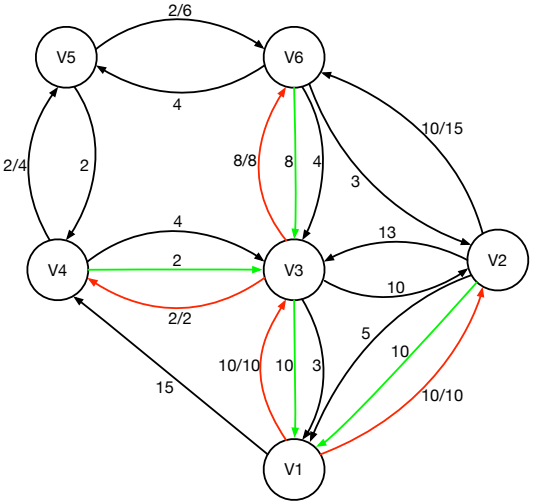
Поиск максимального потока

- Новый поиск дал нам новый путь $V_1 \to V_2 \to V_6$.



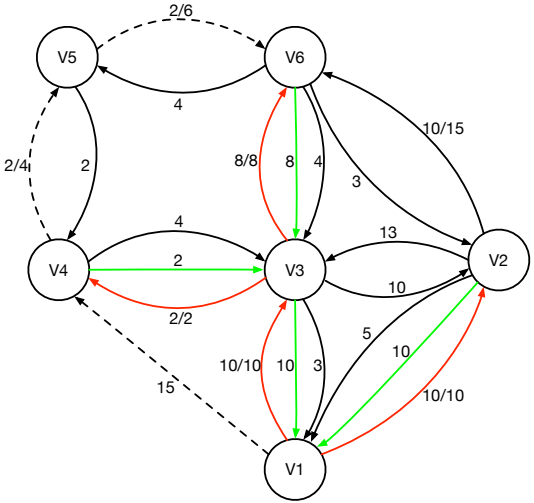
Поиск максимального потока

- Насыщаем рёбра этого пути и добавляем обратные.



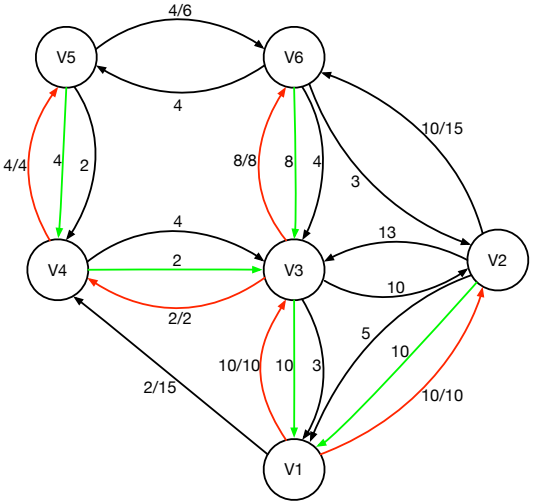
Поиск максимального потока

- Следующий поиск дал нам поток интенсивностью 2.



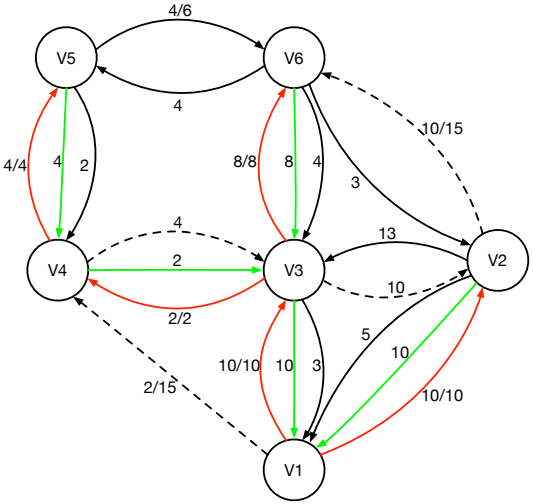
Поиск максимального потока

- Он насытил ребро $V_4 \to V_5$.



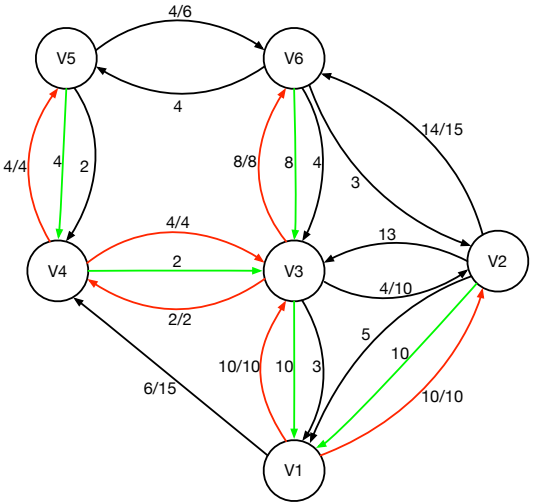
Поиск максимального потока

- Потоки искать всё сложнее, но мы нашли один с интенсивностью 4.



Поиск максимального потока

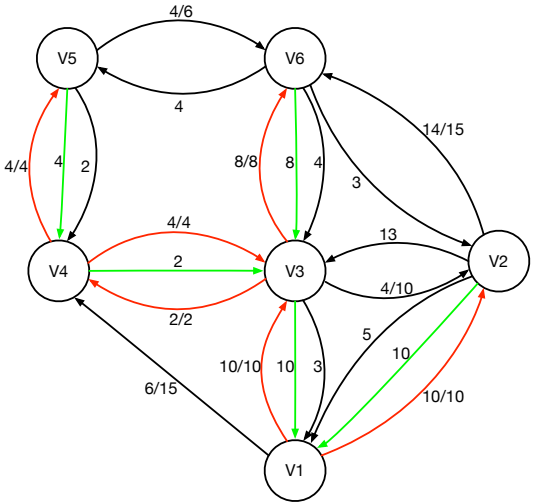
- Мы всё насытили и вот результат:



Максимальный ли это поток? Сейчас он равен 26.

Поиск максимального потока

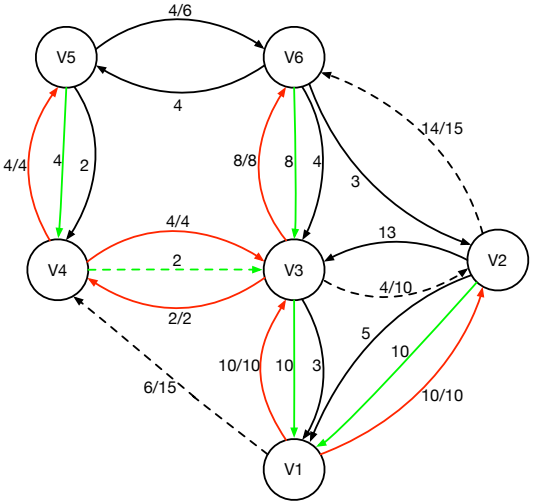
- Мы всё насытили и вот результат:



Максимальный ли это поток? Сейчас он равен 26. Вспомним про обратные рёбра.

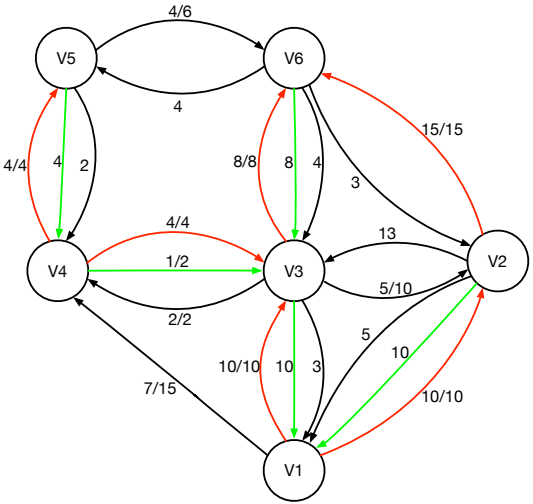
Поиск максимального потока

- Пропустим единицу потока по маршруту $V_1 \to V_4 \to V_3 \to V_2 \to V_6$.



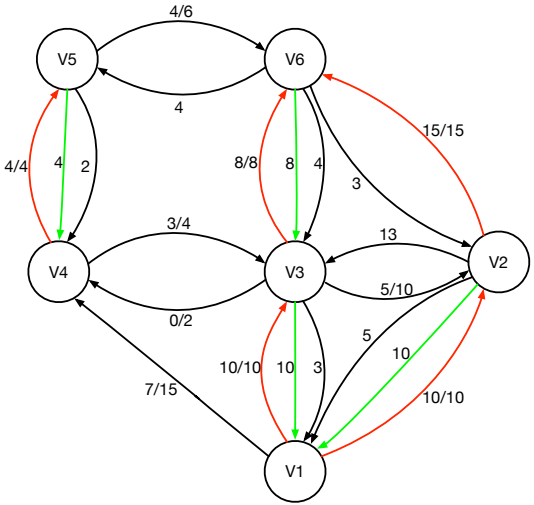
Поиск максимального потока

- Но ведь у нас уже есть две единицы потока из V_3 в V_4.
- Объединим потоки из V_3 в V_4 (пять единиц) и из V_4 в V_3 (две единицы).



Поиск максимального потока

- Больше ничего никуда не добавить — алгоритм завершён.



Из V_1 выходит 27 единиц из 35 возможных, в V_6 приходит 27 из 29 возможных. Что можно расширить, чтобы увеличить пропускную способность?