Мини-задача #34 (2 балла)

Решаем задачу на поиск минимального остовного дерева:

https://classroom.github.com/assignment-invitations/b4a7
21959e995116b46c1274103134a9/status

Используйте для этого алгоритм Прима с оптимизацией через хипы.

Мини-задача #35 (1-2 балла)

Пусть было дерево, в которое добавили одно ребро, сделав его графом. Ваша задача найти некоторое ребро, удалив которое вы снова получите дерево. Если таких ребер несколько - взять последнее из перечисленных во входных данных.

https://leetcode.com/problems/redundant-connection/

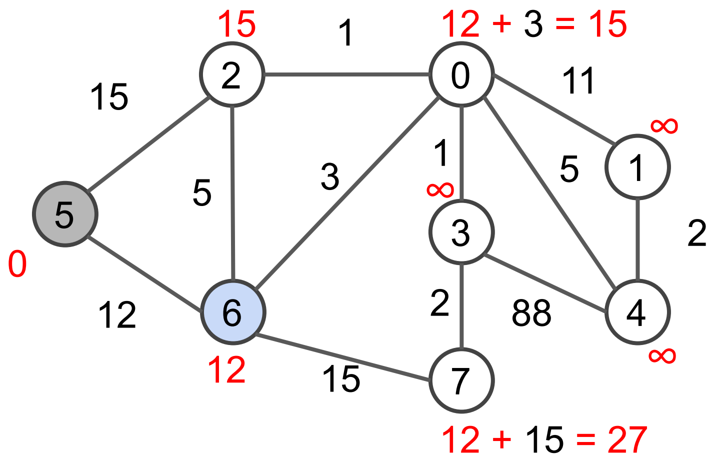
В union-find реализовать оптимизации: либо union by size, либо сжатие путей + объединением по рангам.

За 1 доп. балл решите задачу для ориентированного графа: https://leetcode.com/problems/redundant-connection-ii

Алгоритмы и структуры данных

Жадные алгоритмы на графах, union-find

Алгоритм Дейкстры поиска кратчайших путей



visited =

0	0	0	0	0	1	0	0

dists =

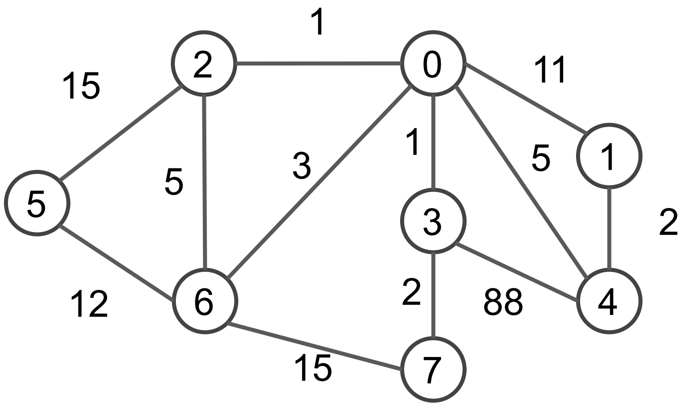
15	∞	15	∞	∞	0	12	27

1. Улучшаем текущее расстояние до соседних с текущей (еще не обработанных) вершинах

2. Пометить текущую, как обработанную

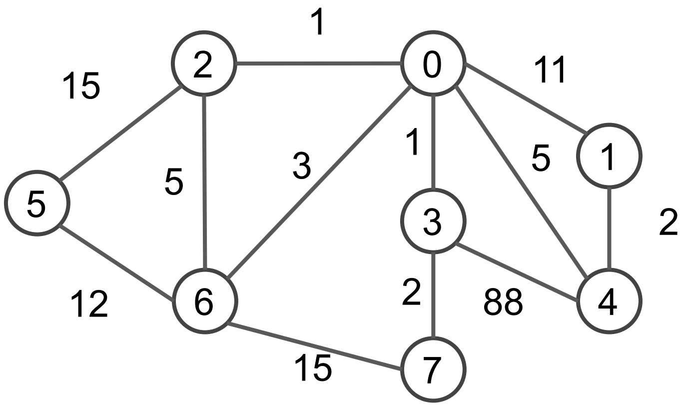
3. В качестве следующей вершины берем ту, в которой еще не были, и у которой минимальный dist

Задача поиска минимального остовного дерева



Дано: связный взвешенный неориентированный граф $G = (V, E)$

Задача поиска минимального остовного дерева

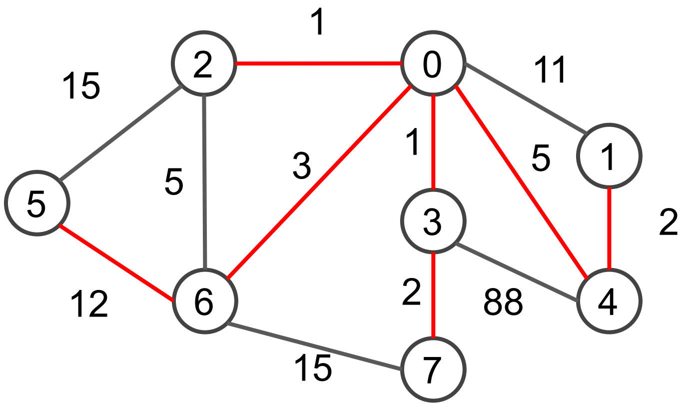


Дано: связный взвешенный неориентированный граф $G = (V, E)$

Найти: подмножество ребер E', такое что граф $G = (V, E')$

1) Является связным и ацикличным
2) Сумма весов его ребер минимальна

Задача поиска минимального остовного дерева

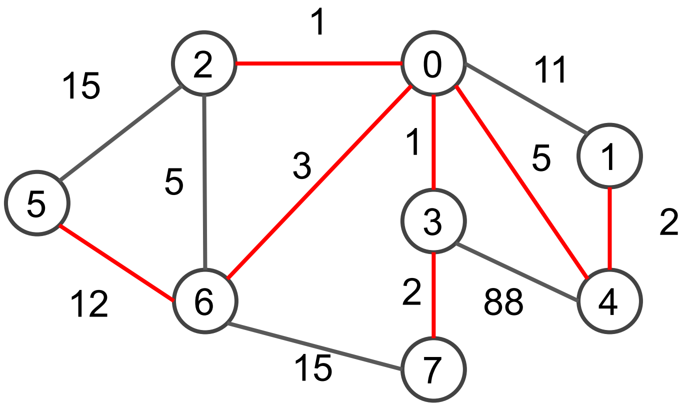


Дано: связный взвешенный неориентированный граф $G = (V, E)$

Найти: подмножество ребер E', такое что граф $G = (V, E')$

1) Является связным и ацикличным
2) Сумма весов его ребер минимальна

Задача поиска минимального остовного дерева



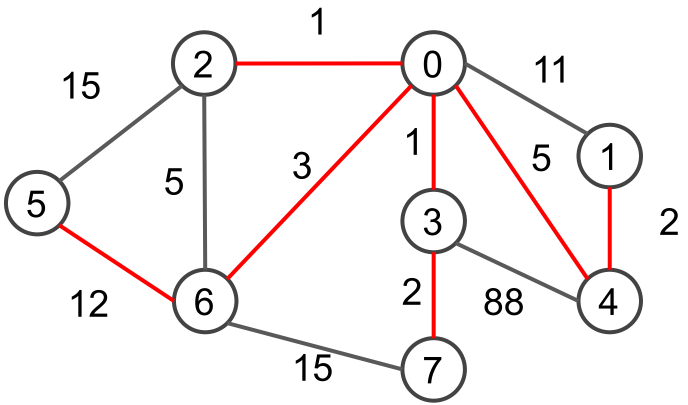
Дано: связный взвешенный неориентированный граф $G = (V, E)$

Найти: подмножество ребер E', такое что граф $G = (V, E')$

1) Является связным и ацикличным
2) Сумма весов его ребер минимальна

Зачем: 1) Экономия ресурсов на дороги/кабели/схемы и т.д.

Задача поиска минимального остовного дерева



Дано: связный взвешенный неориентированный граф $G = (V, E)$

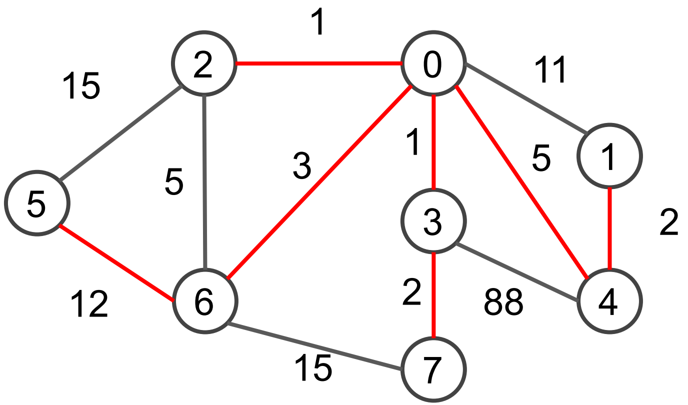
Найти: подмножество ребер E', такое что граф $G = (V, E')$

1) Является связным и ацикличным
2) Сумма весов его ребер минимальна

Зачем: 1) Экономия ресурсов на дороги/кабели/схемы и т.д.

2) Может навести нас на мысли о решении других задач!

Задача поиска минимального остовного дерева



Дано: связный взвешенный неориентированный граф $G = (V, E)$

Найти: подмножество ребер E', такое что граф $G = (V, E')$

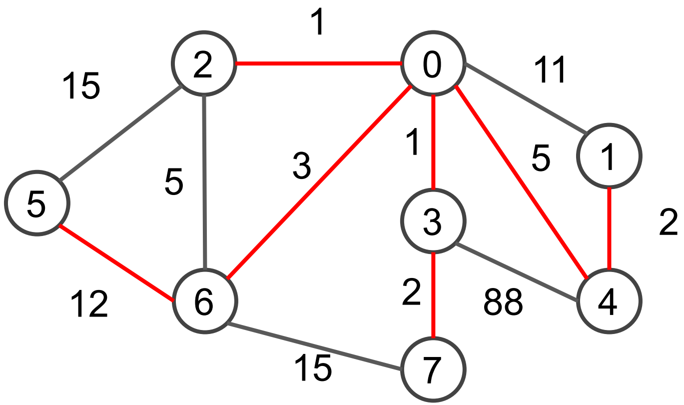
1) Является связным и ацикличным
2) Сумма весов его ребер минимальна

Как: алгоритмы Прима и Краскала*

Зачем: 1) Экономия ресурсов на дороги/кабели/схемы и т.д.

2) Может навести нас на мысли о решении других задач!

Задача поиска минимального остовного дерева



Дано: связный взвешенный неориентированный граф $G = (V, E)$

Найти: подмножество ребер E', такое что граф $G = (V, E')$

1) Является связным и ацикличным
2) Сумма весов его ребер минимальна

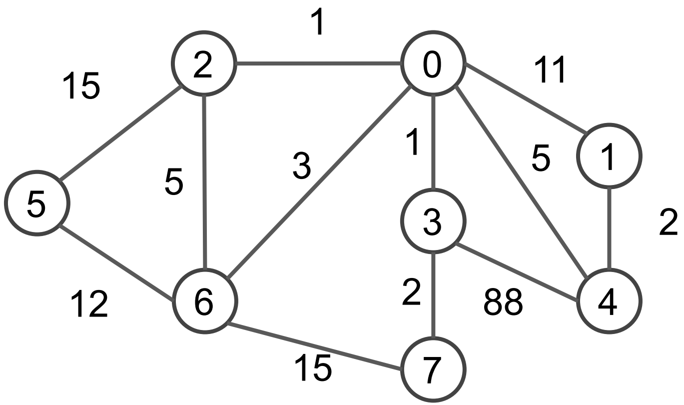
Как: алгоритмы Прима и Краскала*

*Был на дискретке но мы углубимся в union-find.

Зачем: 1) Экономия ресурсов на дороги/кабели/схемы и т.д.

2) Может навести нас на мысли о решении других задач!

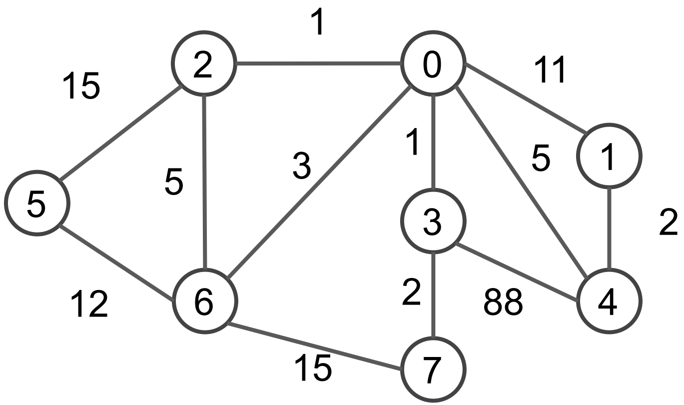
Алгоритм Прима



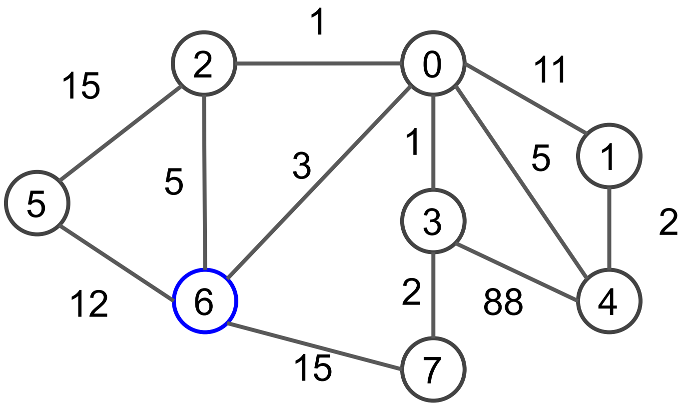
Алгоритм Прима

Строим два множества:

1. X — обработанные вершины
2. T — включенные в остов ребра



Алгоритм Прима



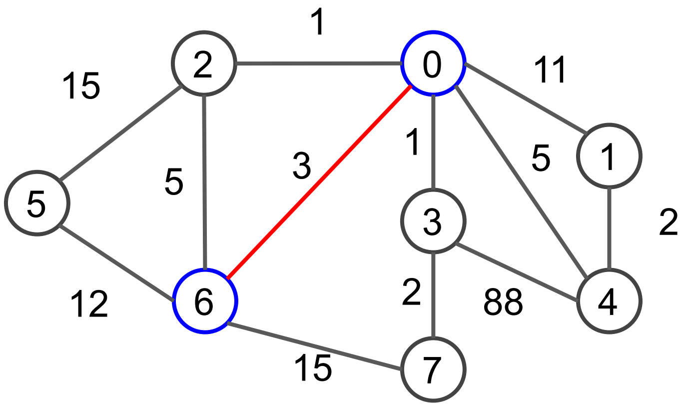
Строим два множества:

1. X — обработанные вершины
2. T — включенные в остов ребра

Алгоритм Прима:

0) Инициализируем X любой вершиной

Алгоритм Прима



Строим два множества:

1. X — обработанные вершины
2. T — включенные в остов ребра

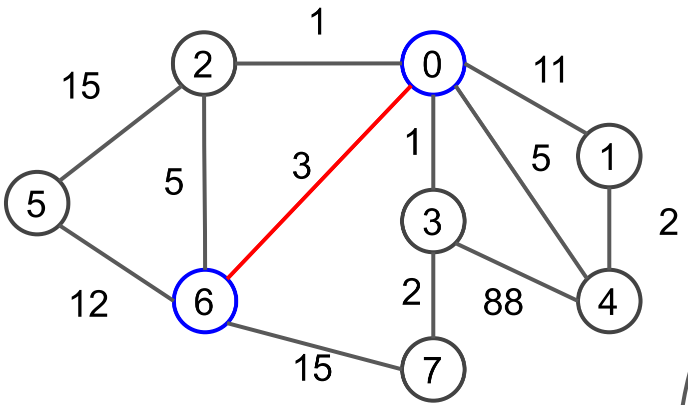
Алгоритм Прима:

0) Инициализируем X любой вершиной

1) Выбираем ребро из X в V\X с наименьшим весом

2) Добавляем его в T, а вершину, куда оно ведет, в X

Алгоритм Прима



Строим два множества:

1. X — обработанные вершины
2. T — включенные в остов ребра

Алгоритм Прима:

0) Инициализируем X любой вершиной

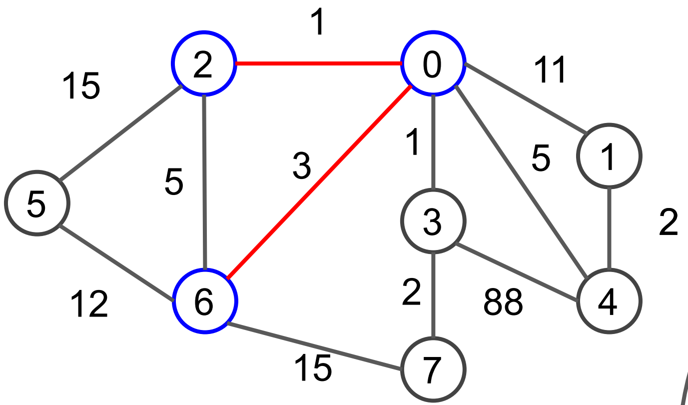
1) Выбираем ребро из X в V\X с наименьшим весом

2) Добавляем его в T, а вершину, куда оно ведет, в X

3) Повторяем, пока X != V

4) Ответ: $G' = (X, T)$

Алгоритм Прима



Строим два множества:

1. X — обработанные вершины
2. T — включенные в остов ребра

Алгоритм Прима:

0) Инициализируем X любой вершиной

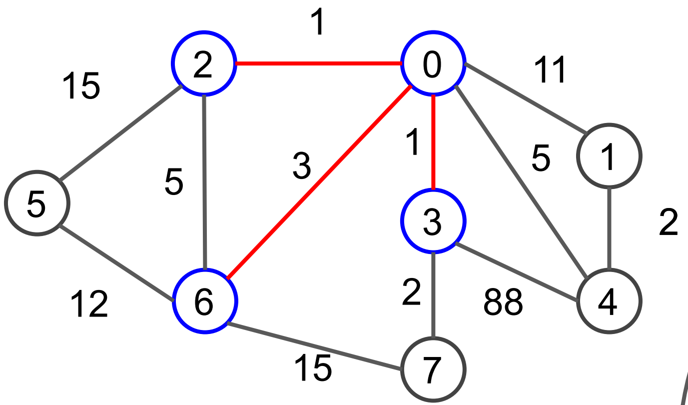
1) Выбираем ребро из X в V\X с наименьшим весом

2) Добавляем его в T, а вершину, куда оно ведет, в X

3) Повторяем, пока X != V

4) Ответ: $G' = (X, T)$

Алгоритм Прима



Строим два множества:

1. X — обработанные вершины
2. T — включенные в остов ребра

Алгоритм Прима:

0) Инициализируем X любой вершиной

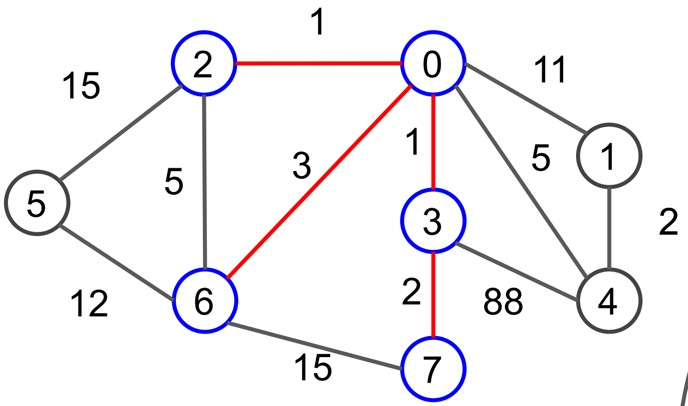
1) Выбираем ребро из X в V\X с наименьшим весом

2) Добавляем его в T, а вершину, куда оно ведет, в X

3) Повторяем, пока X != V

4) Ответ: $G' = (X, T)$

Алгоритм Прима



Строим два множества:

1. X — обработанные вершины
2. T — включенные в остов ребра

Алгоритм Прима:

0) Инициализируем X любой вершиной

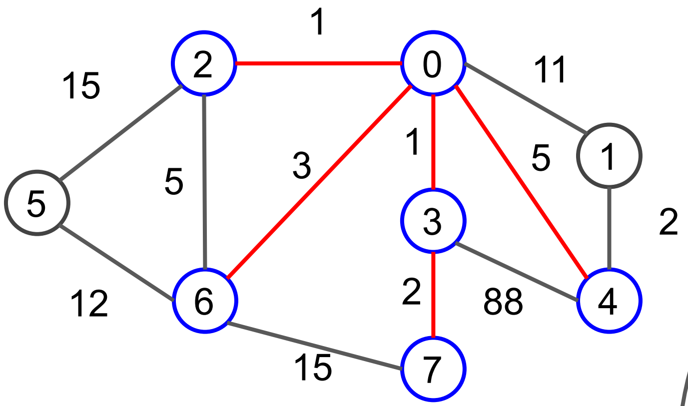
1) Выбираем ребро из X в V\X с наименьшим весом

2) Добавляем его в T, а вершину, куда оно ведет, в X

3) Повторяем, пока X != V

4) Ответ: $G' = (X, T)$

Алгоритм Прима



Строим два множества:

1. X — обработанные вершины
2. T — включенные в остов ребра

Алгоритм Прима:

0) Инициализируем X любой вершиной

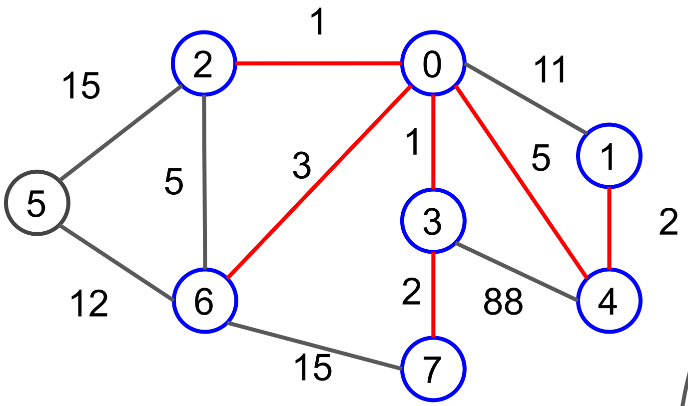
1) Выбираем ребро из X в V\X с наименьшим весом

2) Добавляем его в T, а вершину, куда оно ведет, в X

3) Повторяем, пока X != V

4) Ответ: $G' = (X, T)$

Алгоритм Прима



Строим два множества:

1. X — обработанные вершины
2. T — включенные в остов ребра

Алгоритм Прима:

0) Инициализируем X любой вершиной

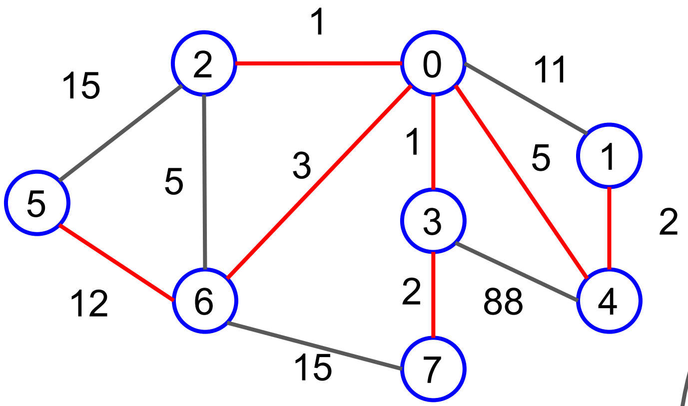
1) Выбираем ребро из X в V\X с наименьшим весом

2) Добавляем его в T, а вершину, куда оно ведет, в X

3) Повторяем, пока X != V

4) Ответ: $G' = (X, T)$

Алгоритм Прима



Строим два множества:

1. X — обработанные вершины
2. T — включенные в остов ребра

Алгоритм Прима:

0) Инициализируем X любой вершиной

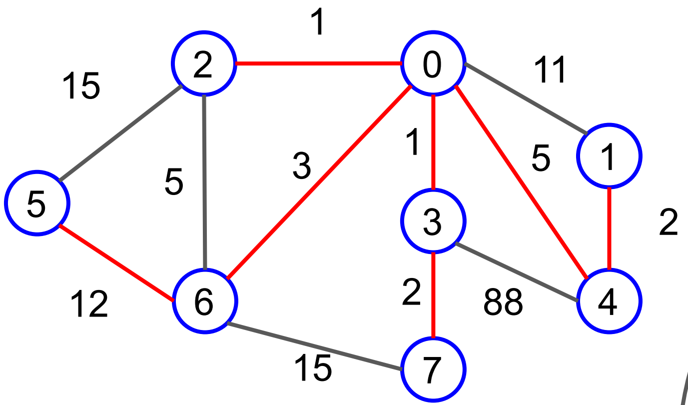
1) Выбираем ребро из X в V\X с наименьшим весом

2) Добавляем его в T, а вершину, куда оно ведет, в X

3) Повторяем, пока X != V

4) Ответ: $G' = (X, T)$

Алгоритм Прима



Это жадный алгоритм, т.к. на каждом шаге выбираем самое дешевое ребро.

Строим два множества:

1. X — обработанные вершины
2. T — включенные в остов ребра

Алгоритм Прима:

0) Инициализируем X любой вершиной

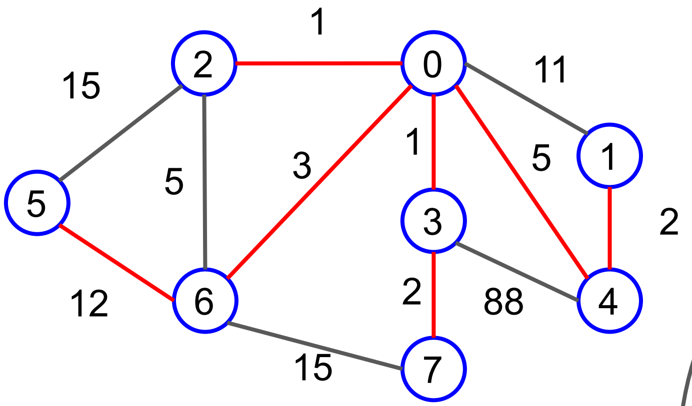
1) Выбираем ребро из X в V\X с наименьшим весом

2) Добавляем его в T, а вершину, куда оно ведет, в X

3) Повторяем, пока X != V

4) Ответ: $G' = (X, T)$

Алгоритм Прима



Строим два множества:

1. X — обработанные вершины
2. T — включенные в остов ребра

Алгоритм Прима:

0) Инициализируем X любой вершиной

1) Выбираем ребро из X в V\X с наименьшим весом

2) Добавляем его в T, а вершину, куда оно ведет, в X

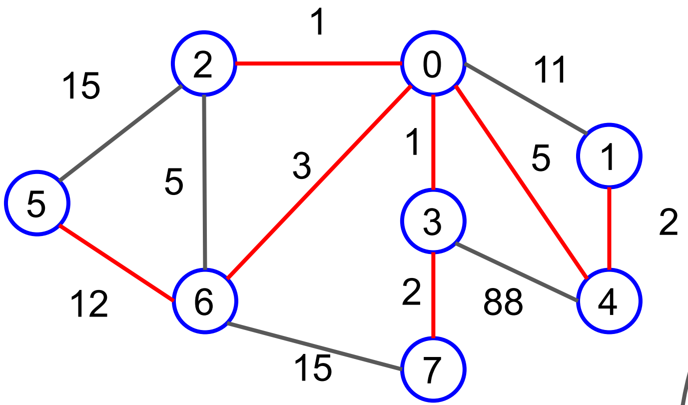
3) Повторяем, пока X != V

4) Ответ: $G' = (X, T)$

Это жадный алгоритм, т.к. на каждом шаге выбираем самое дешевое ребро.

На что похоже?

Алгоритм Прима



Строим два множества:

1. X — обработанные вершины
2. T — включенные в остов ребра

Алгоритм Прима:

0) Инициализируем X любой вершиной

1) Выбираем ребро из X в V\X с наименьшим весом

2) Добавляем его в T, а вершину, куда оно ведет, в X

3) Повторяем, пока X != V

4) Ответ: $G' = (X, T)$

Это жадный алгоритм, т.к. на каждом шаге выбираем самое дешевое ребро.

На что похоже? Брат-близнец алгоритма Дейкстры!

Алгоритм Прима: корректность, часть #1

Почему алгоритм Прима находит хоть какое-то остовное дерево? (пока не говорим про минимальность)

Алгоритм Прима: корректность, часть #1

Почему алгоритм Прима находит хоть какое-то остовное дерево? (пока не говорим про минимальность)

Нужно показать, что полученный граф будет:

а) связным и ацикличным

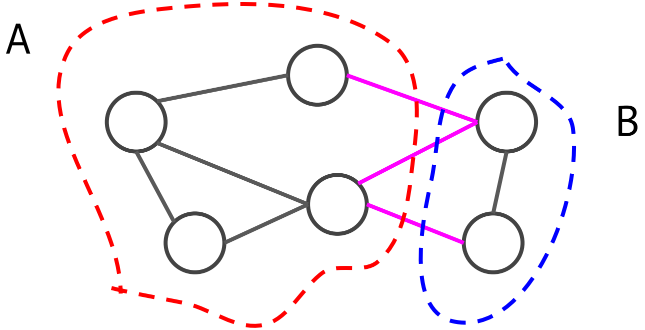
б) содержать все вершины из V

Задача о минимальном разрезе графа

Пусть есть неориентированный граф $G = (V, E)$

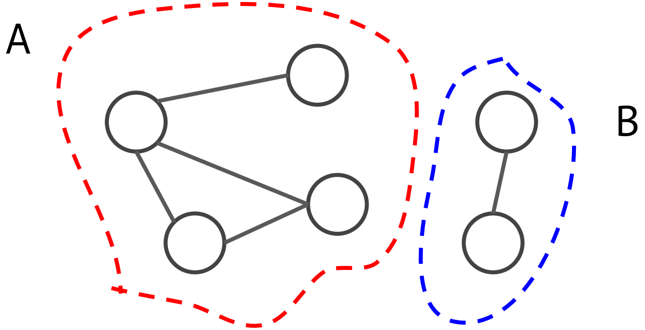
Разрез графа - разбиение V на два непересекающихся непустых множества A и B.

Пересекающие ребра разреза (A,B) - ребра, начинающиеся в A и заканчивающиеся в B.



Тривиальные свойства разрезов

Лемма #1: граф несвязный ⇔ ∃ разрез (A, B) без пересекающих ребер

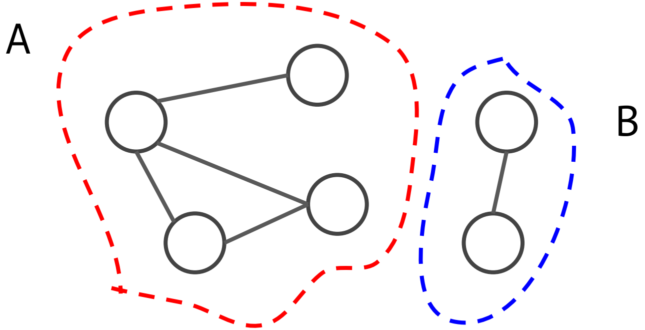


Тривиальные свойства разрезов

Лемма #1: граф несвязный ⇔ ∃ разрез (A, B) без пересекающих ребер

Доказательство:

⇒ граф несвязный, значит есть хотя бы две компоненты связности, возьмем одну из них в качестве A, все остальное в качестве B. Получили искомый разрез.



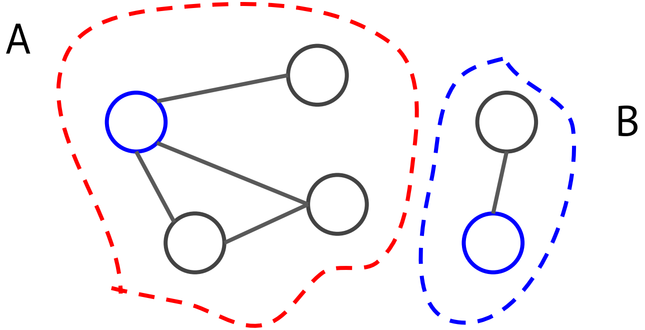
Тривиальные свойства разрезов

Лемма #1: граф несвязный ⇔ ∃ разрез (A, B) без пересекающих ребер

Доказательство:

⇒ граф несвязный, значит есть хотя бы две компоненты связности, возьмем одну из них в качестве A, все остальное в качестве B. Получили искомый разрез.

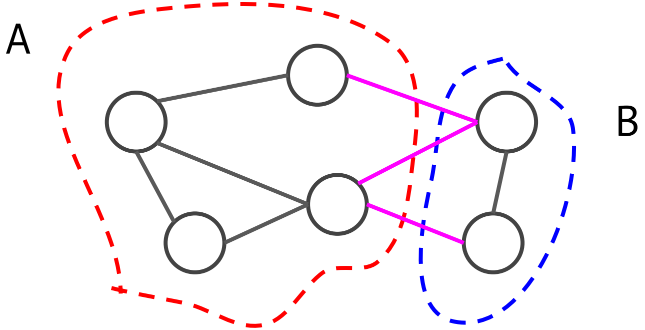
⇐ возьмем одну вершину из A, вторую из B; по условию не может быть пути из одной из этих вершин, в другую, т.к. нет пересекающих ребер



Тривиальные свойства разрезов

Лемма #1: граф несвязный ⇔ ∃ разрез (A, B) без пересекающих ребер

Лемма #2: если в некоторый цикл входило ребро, пересекающее разрез (A, B), то в нем есть и еще одно пересекающее ребро.

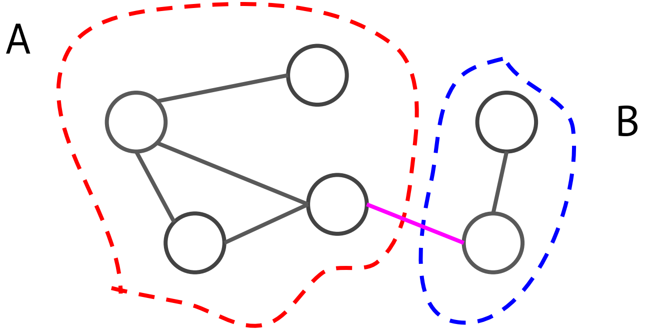


Тривиальные свойства разрезов

Лемма #1: граф несвязный ⇔ ∃ разрез (A, B) без пересекающих ребер

Лемма #2: если в некоторый цикл входило ребро, пересекающее разрез (A, B), то в нем есть и еще одно пересекающее ребро.

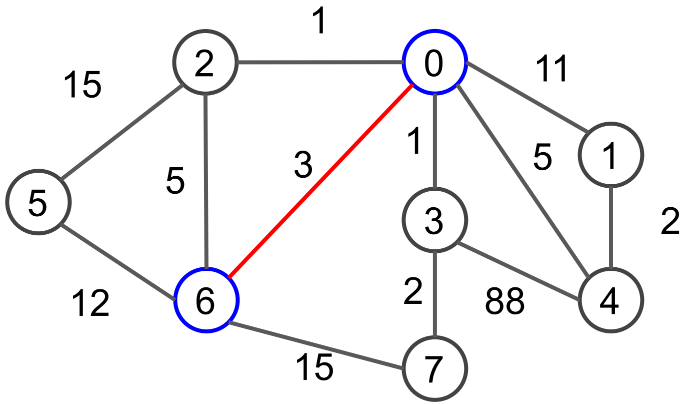
Лемма #3: если есть только одно пересекающее разрез ребро, то оно не входит ни в какой цикл.



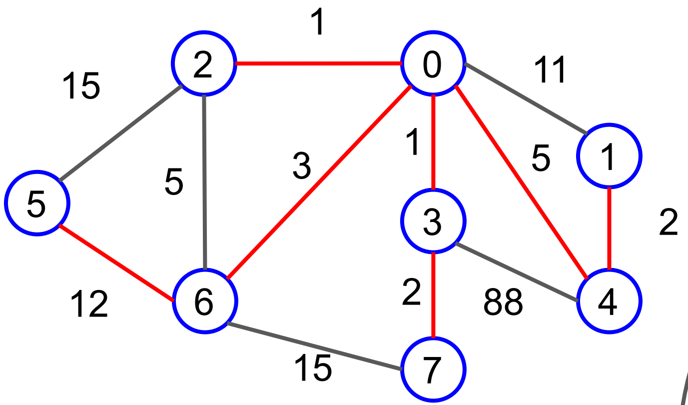
Алгоритм Прима: корректность, часть #1

Утверждение: алгоритм Прима находит остовное дерево в графе

Доказательство:



Алгоритм Прима



Строим два множества:

1. X — обработанные вершины
2. T — включенные в остов ребра

Алгоритм Прима:

0) Инициализируем X любой вершиной

1) Выбираем ребро из X в V\X с наименьшим весом

2) Добавляем его в T, а вершину, куда оно ведет в X

3) Повторяем, пока X != V

4) Ответ: $G' = (X, T)$

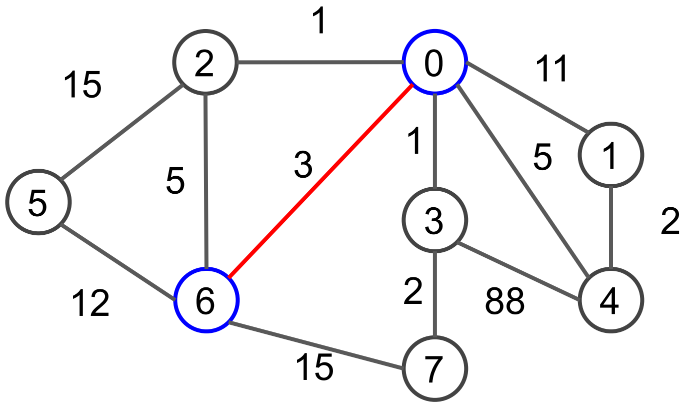
Это жадный алгоритм, т.к. на каждом шаге выбираем самое дешевое ребро.

На что похоже? Брат-близнец алгоритма Дейкстры!

Алгоритм Прима: корректность, часть #1

Утверждение: алгоритм Прима находит остовное дерево в графе

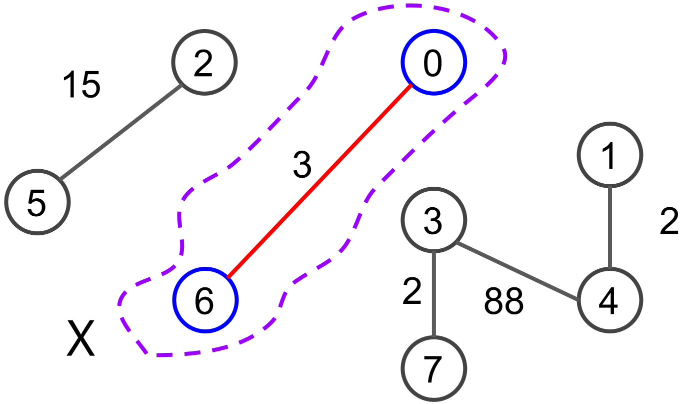
Доказательство: на каждой итерации алгоритм строит разрез из множеств вершин X и V\X. При этом каждый раз выбирается пересекающее ребро, которое добавляется в T.



Алгоритм Прима: корректность, часть #1

Утверждение: алгоритм Прима находит остовное дерево в графе

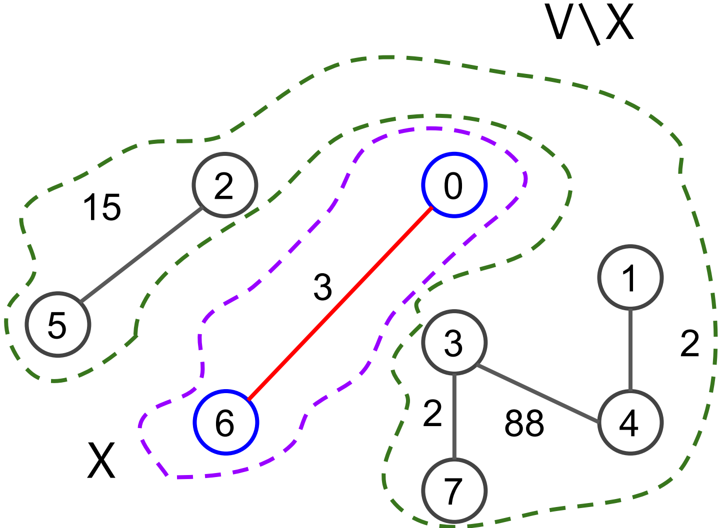
Доказательство: на каждой итерации алгоритм строит разрез из множеств вершин X и V\X. При этом каждый раз выбирается пересекающее ребро, которое добавляется в T.



Алгоритм Прима: корректность, часть #1

Утверждение: алгоритм Прима находит остовное дерево в графе

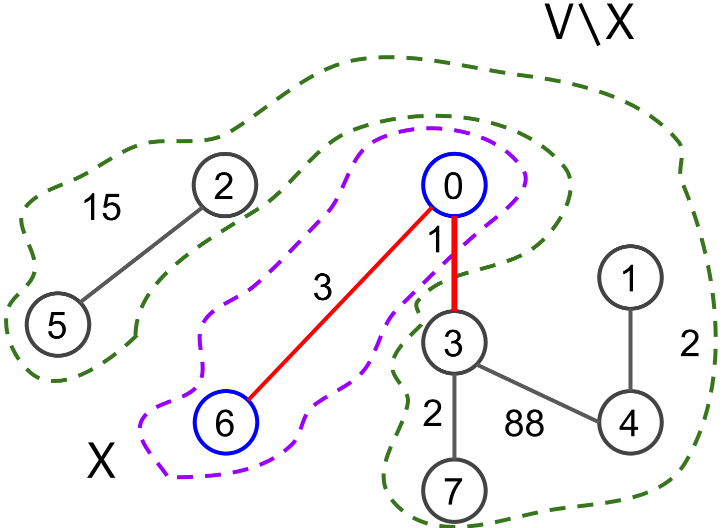
Доказательство: на каждой итерации алгоритм строит разрез из множеств вершин X и V\X. При этом каждый раз выбирается пересекающее ребро, которое добавляется в T.



Алгоритм Прима: корректность, часть #1

Утверждение: алгоритм Прима находит остовное дерево в графе

Доказательство: на каждой итерации алгоритм строит разрез из множеств вершин X и V\X. При этом каждый раз выбирается пересекающее ребро, которое добавляется в T.

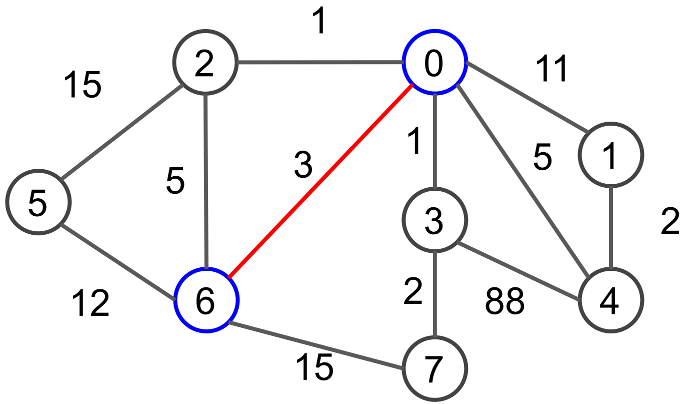


Алгоритм Прима: корректность, часть #1

Утверждение: алгоритм Прима находит остовное дерево в графе

Доказательство: на каждой итерации алгоритм строит разрез из множеств вершин X и V\X. При этом каждый раз выбирается пересекающее ребро, которое добавляется в T.

1) Всегда в конце получаем, что X = V.

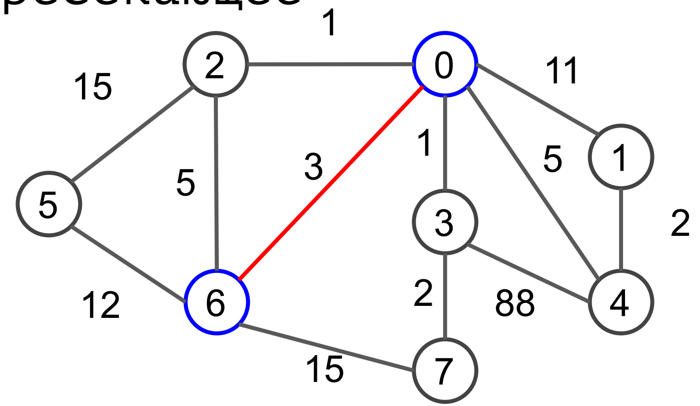


Алгоритм Прима: корректность, часть #1

Утверждение: алгоритм Прима находит остовное дерево в графе

Доказательство: на каждой итерации алгоритм строит разрез из множеств вершин X и V\X. При этом каждый раз выбирается пересекающее ребро, которое добавляется в T.

1) Всегда в конце получаем, что X = V. Иначе мы бы на каком-то шаге не смогли взять пересекающее ребро => изначальный граф был бы несвязным по лемме #1



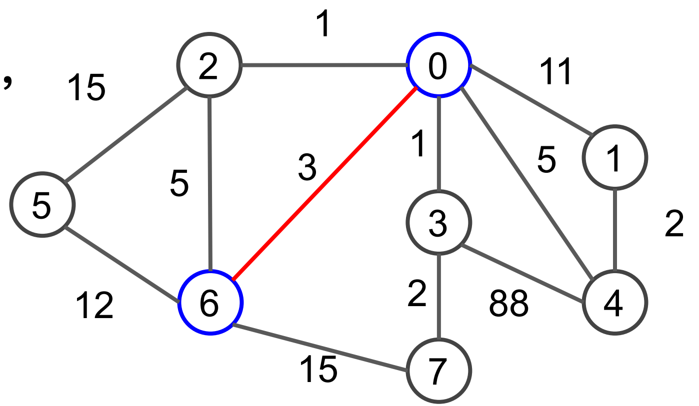
Алгоритм Прима: корректность, часть #1

Утверждение: алгоритм Прима находит остовное дерево в графе

Доказательство: на каждой итерации алгоритм строит разрез из множеств вершин X и V\X. При этом каждый раз выбирается пересекающее ребро, которое добавляется в T.

1) Всегда в конце получаем, что X = V.

2) Аналогично по построению получаем, что что новый граф связный.



Алгоритм Прима: корректность, часть #1

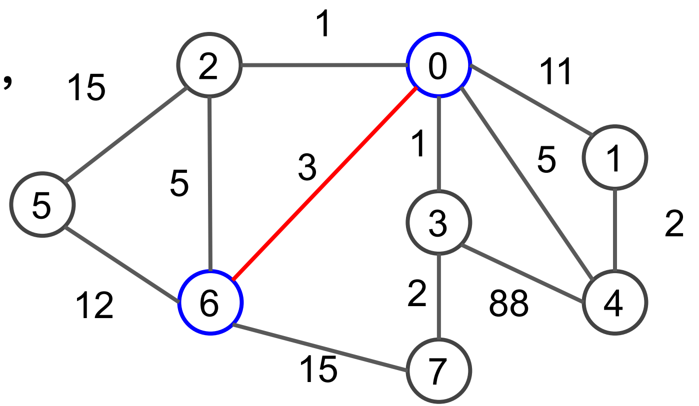
Утверждение: алгоритм Прима находит остовное дерево в графе

Доказательство: на каждой итерации алгоритм строит разрез из множеств вершин X и V\X. При этом каждый раз выбирается пересекающее ребро, которое добавляется в T.

1) Всегда в конце получаем, что X = V.

2) Аналогично по построению получаем, что что новый граф связный.

3) А циклов почему нет?



Алгоритм Прима: корректность, часть #1

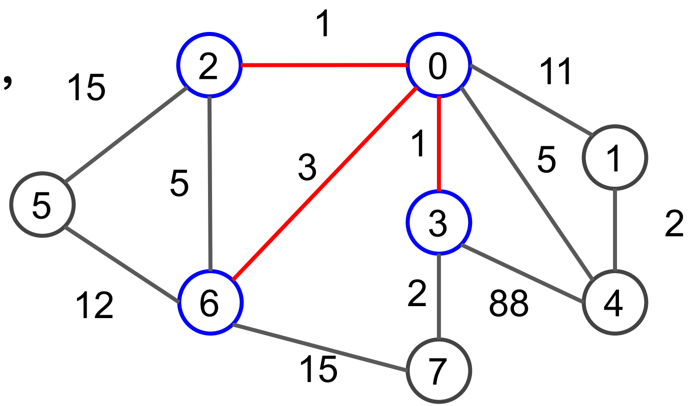
Утверждение: алгоритм Прима находит остовное дерево в графе

Доказательство: на каждой итерации алгоритм строит разрез из множеств вершин X и V\X. При этом каждый раз выбирается пересекающее ребро, которое добавляется в T.

1) Всегда в конце получаем, что X = V.

2) Аналогично по построению получаем, что что новый граф связный.

3) А циклов почему нет?



Алгоритм Прима: корректность, часть #1

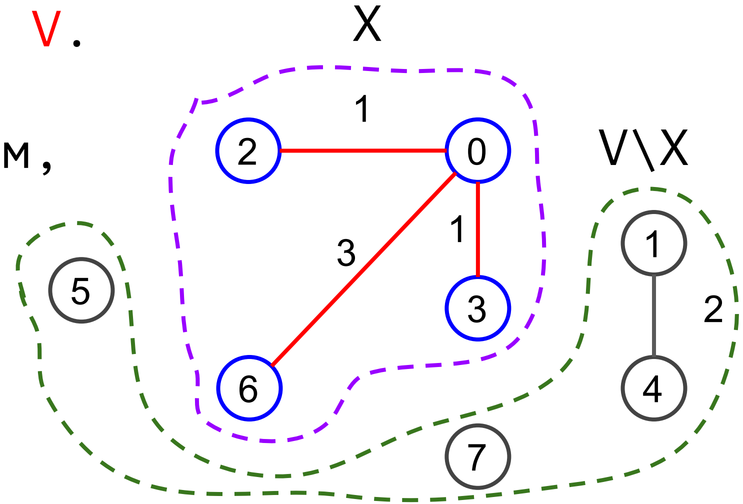
Утверждение: алгоритм Прима находит остовное дерево в графе

Доказательство: на каждой итерации алгоритм строит разрез из множеств вершин X и V\X. При этом каждый раз выбирается пересекающее ребро, которое добавляется в T.

1) Всегда в конце получаем, что X = V.

2) Аналогично по построению получаем, что что новый граф связный.

3) А циклов почему нет?



Алгоритм Прима: корректность, часть #1

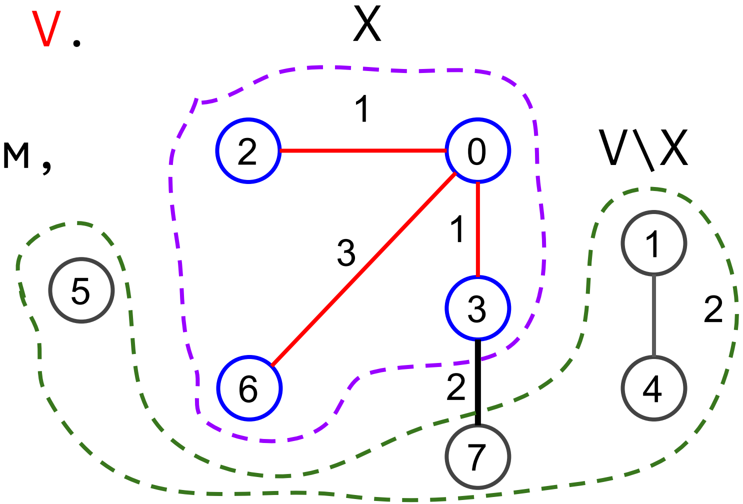
Утверждение: алгоритм Прима находит остовное дерево в графе

Доказательство: на каждой итерации алгоритм строит разрез из множеств вершин X и V\X. При этом каждый раз выбирается пересекающее ребро, которое добавляется в T.

1) Всегда в конце получаем, что X = V.

2) Аналогично по построению получаем, что что новый граф связный.

3) А циклов почему нет? По построению это первое ребро из разреза, которое мы добавляем

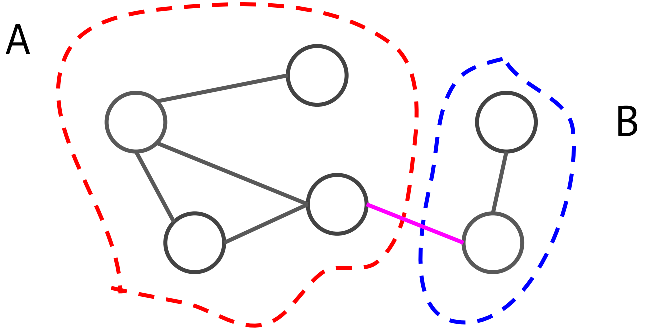


Тривиальные свойства разрезов

Лемма #1: граф несвязный ⇔ ∃ разрез (A, B) без пересекающих ребер

Лемма #2: если в некоторый цикл входит ребро, пересекающее разрез (A, B), то в нем есть и еще одно пересекающее ребро.

Лемма #3: если есть только одно пересекающее разрез ребро, то оно не входит ни в какой цикл.



Алгоритм Прима: корректность, часть #1

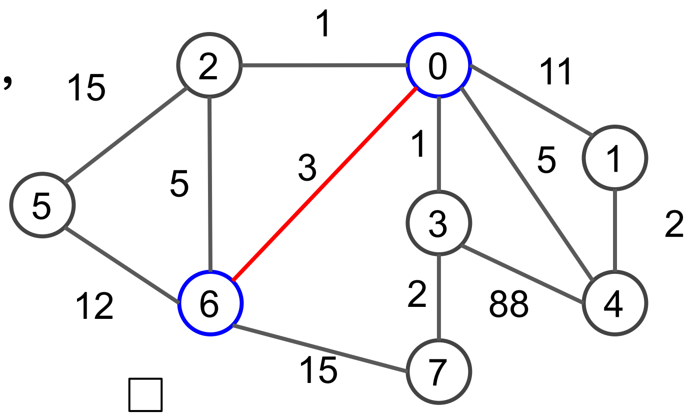
Утверждение: алгоритм Прима находит остовное дерево в графе

Доказательство: на каждой итерации алгоритм строит разрез из множеств вершин X и V\X. При этом каждый раз выбирается пересекающее ребро, которое добавляется в T.

1) Всегда в конце получаем, что X = V.

2) Аналогично по построению получаем, что что новый граф связный.

3) А циклов нет по лемме 3, т.к. на каждой итерации добавляли в T первое пересекающее ребро



□

Алгоритм Прима: корректность, часть #2

Утверждение: алгоритм Прима находит минимальное остовное дерево в графе.

Алгоритм Прима: корректность, часть #2

Утверждение: алгоритм Прима находит минимальное остовное дерево в графе.

Лемма #4 (свойство разреза): пусть есть некоторый разрез (A, B). Если ребро e - кратчайшее из пересекающих этот разрез ребер, то оно обязательно входит в минимальное остовное дерево.

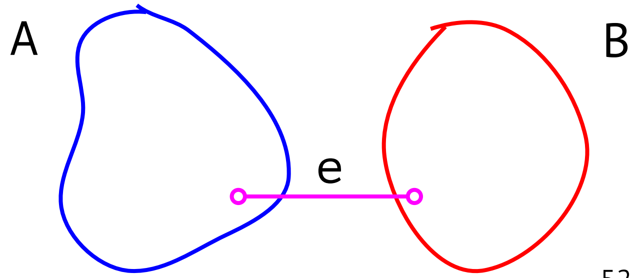
Доказательство: в курсе ДМТА, стр. 43 в методичке.

Лемма #4 (свойство разреза): пусть есть некоторый разрез (A, B). Если ребро e - кратчайшее из пересекающих этот разрез ребер, то оно обязательно входит в минимальное остовное дерево.

Доказательство:

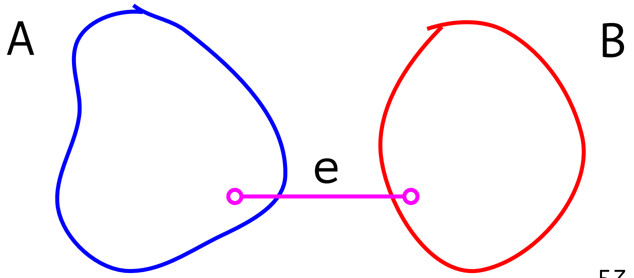
Лемма #4 (свойство разреза): пусть есть некоторый разрез (A, B). Если ребро e - кратчайшее из пересекающих этот разрез ребер, то оно обязательно входит в минимальное остовное дерево.

Доказательство:



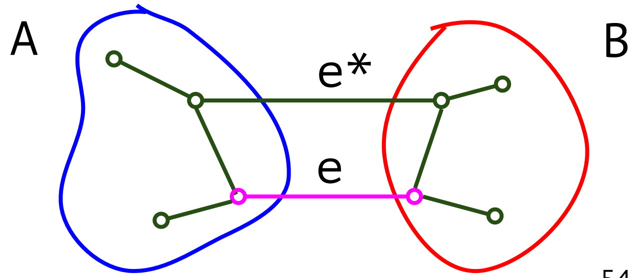
Лемма #4 (свойство разреза): пусть есть некоторый разрез (A, B). Если ребро e - кратчайшее из пересекающих этот разрез ребер, то оно обязательно входит в минимальное остовное дерево.

Доказательство: предположим, что это не так => e не входит в минимальное остовное дерево (обозначим его **T**).



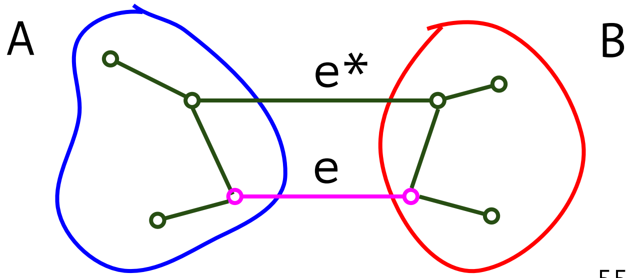
Лемма #4 (свойство разреза): пусть есть некоторый разрез (A, B). Если ребро e - кратчайшее из пересекающих этот разрез ребер, то оно обязательно входит в минимальное остовное дерево.

Доказательство: предположим, что это не так => e не входит в минимальное остовное дерево (обозначим его **T**). Тогда существует e* ∈ T, пересекающее (A, B).



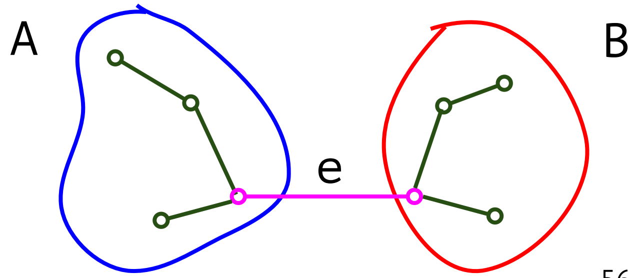
Лемма #4 (свойство разреза): пусть есть некоторый разрез (A, B). Если ребро e - кратчайшее из пересекающих этот разрез ребер, то оно обязательно входит в минимальное остовное дерево.

Доказательство: предположим, что это не так => e не входит в минимальное остовное дерево (обозначим его **T**). Тогда существует e* ∈ T, пересекающее (A, B). Почему?



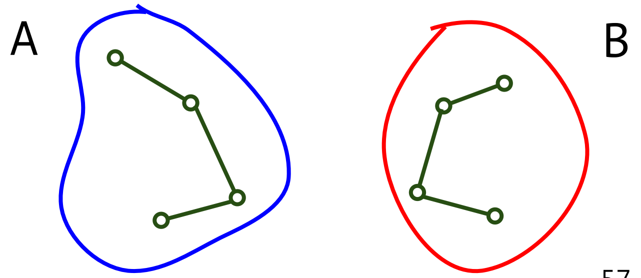
Лемма #4 (свойство разреза): пусть есть некоторый разрез (A, B). Если ребро e - кратчайшее из пересекающих этот разрез ребер, то оно обязательно входит в минимальное остовное дерево.

Доказательство: предположим, что это не так => e не входит в минимальное остовное дерево (обозначим его **T**). Тогда существует e* ∈ T, пересекающее (A, B). Почему? Иначе **T** не будет связным, ведь существует разрез, который не пересекает не одно ребро!



Лемма #4 (свойство разреза): пусть есть некоторый разрез (A, B). Если ребро e - кратчайшее из пересекающих этот разрез ребер, то оно обязательно входит в минимальное остовное дерево.

Доказательство: предположим, что это не так => e не входит в минимальное остовное дерево (обозначим его **T**). Тогда существует e* ∈ T, пересекающее (A, B). Почему? Иначе **T** не будет связным, ведь существует разрез, который не пересекает не одно ребро!

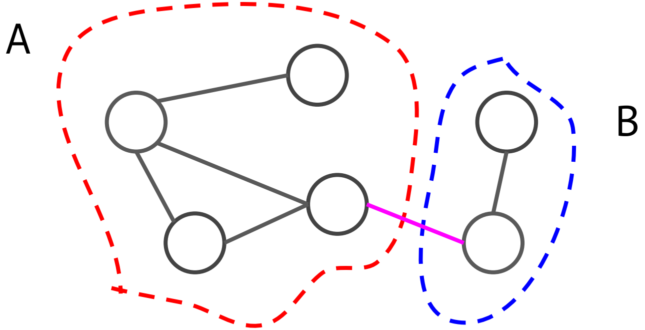


Тривиальные свойства разрезов

Лемма #1: граф несвязный ⇔ ∃ разрез (A, B) без пересекающих ребер

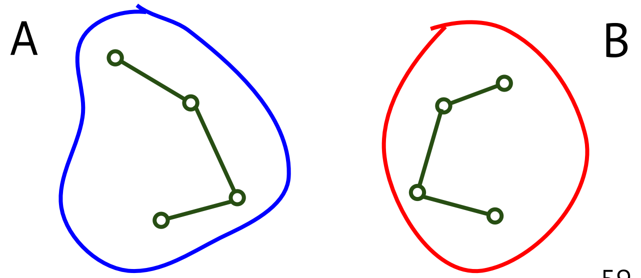
Лемма #2: если в некоторый цикл входил ребро, пересекающее разрез (A, B), то в нем есть и еще одно пересекающее ребро.

Лемма #3: если есть только одно пересекающее разрез ребро, то оно не входит ни в какой цикл.



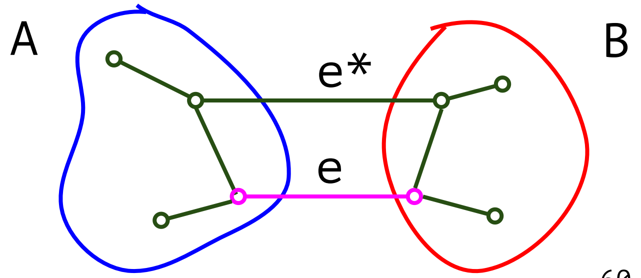
Лемма #4 (свойство разреза): пусть есть некоторый разрез (A, B). Если ребро e - кратчайшее из пересекающих этот разрез ребер, то оно обязательно входит в минимальное остовное дерево.

Доказательство: предположим, что это не так => e не входит в минимальное остовное дерево (обозначим его **T**). Тогда существует e* ∈ T, пересекающее (A, B).



Лемма #4 (свойство разреза): пусть есть некоторый разрез (A, B). Если ребро e - кратчайшее из пересекающих этот разрез ребер, то оно обязательно входит в минимальное остовное дерево.

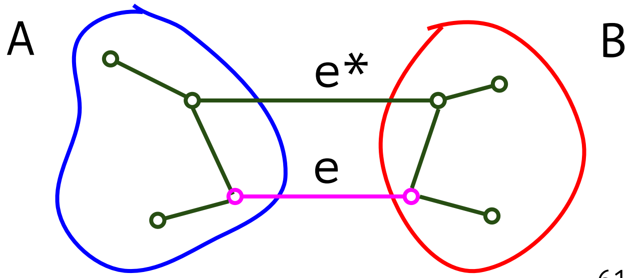
Доказательство: предположим, что это не так => e не входит в минимальное остовное дерево (обозначим его **T**). Тогда существует $e^* \in T$, пересекающее (A, B). Заметим, что $|e^*| > |e|$ (будем считать, что все ребра имеют разный вес).



Лемма #4 (свойство разреза): пусть есть некоторый разрез (A, B). Если ребро e - кратчайшее из пересекающих этот разрез ребер, то оно обязательно входит в минимальное остовное дерево.

Доказательство: предположим, что это не так => e не входит в минимальное остовное дерево (обозначим его **T**). Тогда существует $e^* \in T$, пересекающее (A, B). Заметим, что $|e^*| > |e|$.

Далее попробуем взять $T^* = T - \{e^*\} + \{e\}$

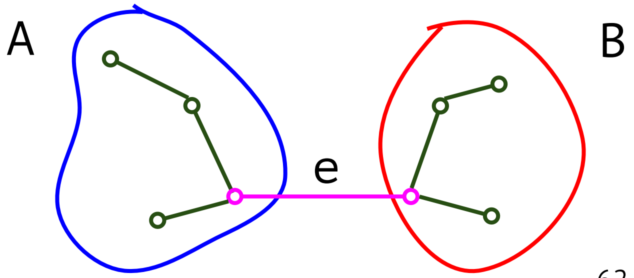


Лемма #4 (свойство разреза): пусть есть некоторый разрез (A, B). Если ребро e - кратчайшее из пересекающих этот разрез ребер, то оно обязательно входит в минимальное остовное дерево.

Доказательство: предположим, что это не так => e не входит в минимальное остовное дерево (обозначим его **T**). Тогда существует $e^* \in T$, пересекающее (A, B). Заметим, что $|e^*| > |e|$.

Далее попробуем взять $T^* = T - \{e^*\} + \{e\}$

Конкретно на этом примере: получаем новое остовное дерево, суммарный вес которого меньше, чем вес T.

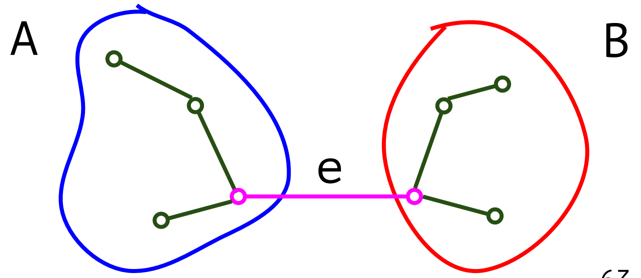


Лемма #4 (свойство разреза): пусть есть некоторый разрез (A, B). Если ребро e - кратчайшее из пересекающих этот разрез ребер, то оно обязательно входит в минимальное остовное дерево.

Доказательство: предположим, что это не так => e не входит в минимальное остовное дерево (обозначим его **T**). Тогда существует e* ∈ T, пересекающее (A, B). Заметим, что |e*| > |e|.

Далее попробуем взять T* = T - {e*} + {e}

Конкретно на этом примере: получаем новое остовное дерево T*, суммарный вес которого меньше, чем вес T.



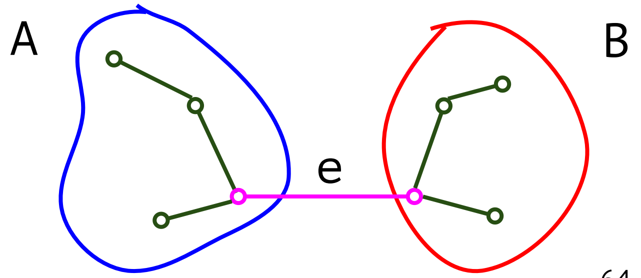
Лемма #4 (свойство разреза): пусть есть некоторый разрез (A, B). Если ребро e - кратчайшее из пересекающих этот разрез ребер, то оно обязательно входит в минимальное остовное дерево.

Доказательство: предположим, что это не так => e не входит в минимальное остовное дерево (обозначим его **T**). Тогда существует $e^* \in T$, пересекающее (A, B). Заметим, что $|e^*| > |e|$.

Далее попробуем взять $T^* = T - \{e^*\} + \{e\}$

Конкретно на этом примере: получаем новое остовное дерево T*, суммарный вес которого меньше, чем вес T.

Но всегда ли T* - это минимальное остовное дерево?



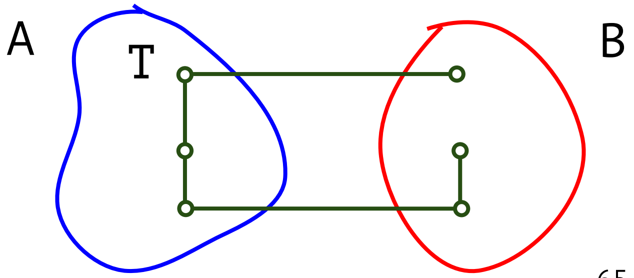
Лемма #4 (свойство разреза): пусть есть некоторый разрез (A, B). Если ребро e - кратчайшее из пересекающих этот разрез ребер, то оно обязательно входит в минимальное остовное дерево.

Доказательство: предположим, что это не так => e не входит в минимальное остовное дерево (обозначим его **T**). Тогда существует e* ∈ T, пересекающее (A, B). Заметим, что |e*| > |e|.

Далее попробуем взять T* = T - {e*} + {e}

Конкретно на этом примере: получаем новое остовное дерево T*, суммарный вес которого меньше, чем вес T.

Но всегда ли T* - это минимальное остовное дерево?



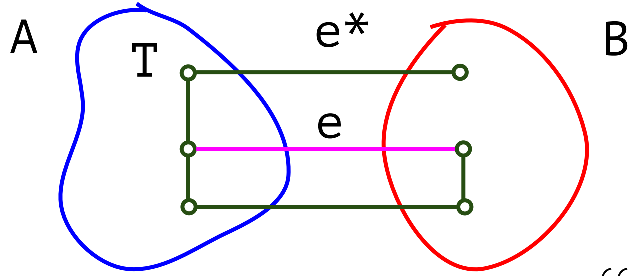
Лемма #4 (свойство разреза): пусть есть некоторый разрез (A, B). Если ребро e - кратчайшее из пересекающих этот разрез ребер, то оно обязательно входит в минимальное остовное дерево.

Доказательство: предположим, что это не так => e не входит в минимальное остовное дерево (обозначим его **T**). Тогда существует e* ∈ T, пересекающее (A, B). Заметим, что |e*| > |e|.

Далее попробуем взять T* = T - {e*} + {e}

~~Конкретно на этом примере: получаем новое остовное дерево T*, суммарный вес которого меньше, чем вес T.~~

Но всегда ли T* - это минимальное остовное дерево?



Лемма #4 (свойство разреза): пусть есть некоторый разрез (A, B). Если ребро e - кратчайшее из пересекающих этот разрез ребер, то оно обязательно входит в минимальное остовное дерево.

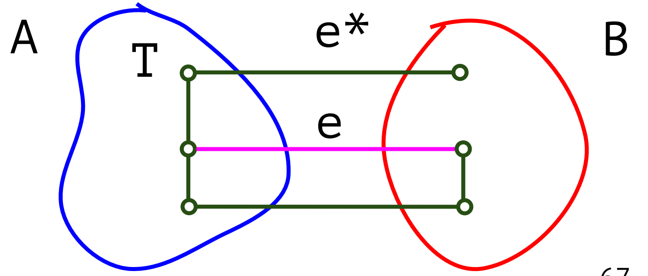
Доказательство: предположим, что это не так => e не входит в минимальное остовное дерево (обозначим его **T**). Тогда существует e* ∈ T, пересекающее (A, B). Заметим, что |e*| > |e|.

Далее попробуем взять T* = T - {e*} + {e}

~~Конкретно на этом примере: получаем новое остовное дерево T*, суммарный вес которого меньше, чем вес T.~~

Но всегда ли T* - это минимальное остовное дерево?

Нет! Замена e* на e может создать цикл!



Лемма #4 (свойство разреза): пусть есть некоторый разрез (A, B). Если ребро e - кратчайшее из пересекающих этот разрез ребер, то оно обязательно входит в минимальное остовное дерево.

Доказательство: предположим, что это не так => e не входит в минимальное остовное дерево (обозначим его **T**). Тогда существует $e^* \in T$, пересекающее (A, B). Заметим, что $|e^*| > |e|$.

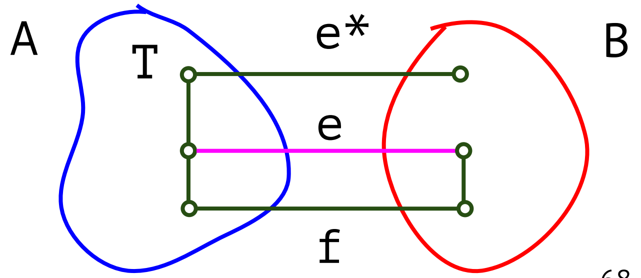
Далее попробуем взять $T^* = T - \{e^*\} + \{e\}$

~~Конкретно на этом примере: получаем новое остовное дерево T*, суммарный вес которого меньше, чем вес T.~~

Но всегда ли T* - это минимальное остовное дерево?

Нет! Замена e* на e может создать цикл!

Но есть и хорошая новость: рассмотрим ребро f. Что можно о нем сказать?



Лемма #4 (свойство разреза): пусть есть некоторый разрез (A, B). Если ребро e - кратчайшее из пересекающих этот разрез ребер, то оно обязательно входит в минимальное остовное дерево.

Доказательство: предположим, что это не так => e не входит в минимальное остовное дерево (обозначим его **T**). Тогда существует $e^* \in T$, пересекающее (A, B). Заметим, что $|e^*| > |e|$.

Далее попробуем взять $T^* = T - \{e^*\} + \{e\}$

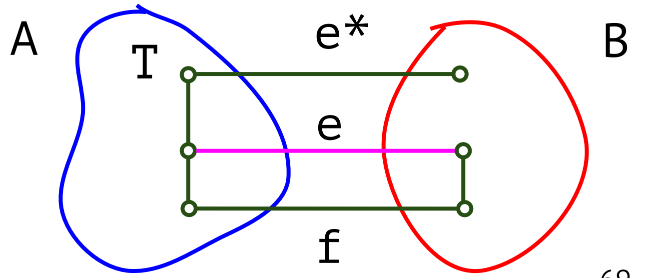
~~Конкретно на этом примере: получаем новое остовное дерево T*, суммарный вес которого меньше, чем вес T.~~

Но всегда ли T* - это минимальное остовное дерево?

Нет! Замена e* на e может создать цикл!

Но есть и хорошая новость: рассмотрим ребро f. Что можно о нем сказать?

Оно существует по лемме 2

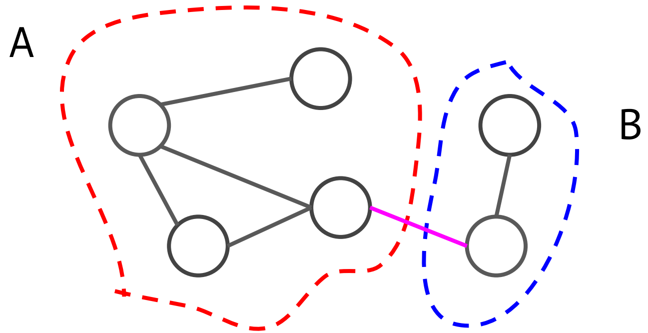


Тривиальные свойства разрезов

Лемма #1: граф несвязный ⇔ ∃ разрез (A, B) без пересекающих ребер

Лемма #2: если в некоторый цикл входит ребро, пересекающее разрез (A, B), то в нем есть и еще одно пересекающее ребро.

Лемма #3: если есть только одно пересекающее разрез ребро, то оно не входит ни в какой цикл.



Лемма #4 (свойство разреза): пусть есть некоторый разрез (A, B). Если ребро e - кратчайшее из пересекающих этот разрез ребер, то оно обязательно входит в минимальное остовное дерево.

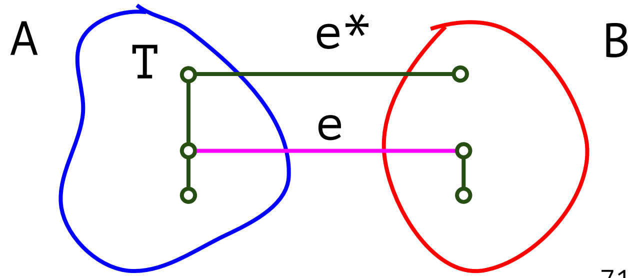
Доказательство: предположим, что это не так => e не входит в минимальное остовное дерево (обозначим его **T**). Тогда существует e* ∈ T, пересекающее (A, B). Заметим, что |e*| > |e|.

Далее попробуем взять T* = T - {e*} + {e}

~~Конкретно на этом примере: получаем новое остовное дерево T*, суммарный вес которого меньше, чем вес T.~~

Но всегда ли T* - это минимальное остовное дерево?

Нет! Замена e* на e может создать цикл!



Но есть и хорошая новость: рассмотрим ребро f. Что можно о нем сказать?

T** = T - {f} + {e} - это МОД.

Лемма #4 (свойство разреза): пусть есть некоторый разрез (A, B). Если ребро e - кратчайшее из пересекающих этот разрез ребер, то оно обязательно входит в минимальное остовное дерево.

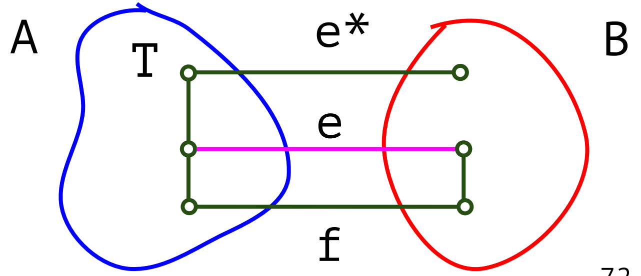
Доказательство: предположим, что это не так => e не входит в минимальное остовное дерево (обозначим его **T**). Тогда существует $e^* \in T$, пересекающее (A, B). Заметим, что $|e^*| > |e|$.

Далее попробуем взять $T^* = T - \{e^*\} + \{e\}$. Но всегда ли T^* - это минимальное остовное дерево? Нет! Замена e^* на e может создать цикл!

Но есть и хорошая новость: рассмотрим ребро f. Что можно о нем сказать?

$T^{**} = T - \{f\} + \{e\}$ - это МОД.

Добавление e создавало цикл, где ровно два ребра пересекали разрез (иначе бы цикл уже был, а мы добавляли в T).



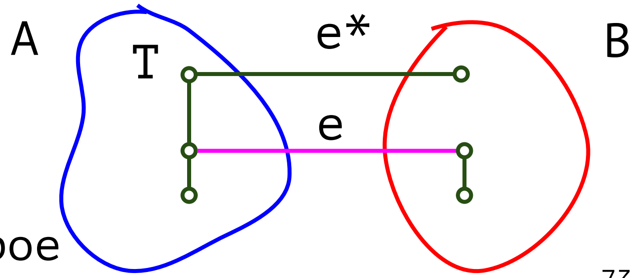
Лемма #4 (свойство разреза): пусть есть некоторый разрез (A, B). Если ребро e - кратчайшее из пересекающих этот разрез ребер, то оно обязательно входит в минимальное остовное дерево.

Доказательство: предположим, что это не так => e не входит в минимальное остовное дерево (обозначим его **T**). Тогда существует $e^* \in T$, пересекающее (A, B). Заметим, что $|e^*| > |e|$.

Далее попробуем взять $T^* = T - \{e^*\} + \{e\}$. Но всегда ли T^* - это минимальное остовное дерево? Нет! Замена e^* на e может создать цикл!

Но есть и хорошая новость: рассмотрим ребро f. Что можно о нем сказать?

$T^{**} = T - \{f\} + \{e\}$ - это МОД.



Добавление e создавало цикл, где ровно два ребра пересекали разрез (иначе бы цикл уже был, а мы добавляли в T). Значит, убрав второе ребро, мы избавимся от цикла (3 лемма).

Лемма #4 (свойство разреза): пусть есть некоторый разрез (A, B). Если ребро e - кратчайшее из пересекающих этот разрез ребер, то оно обязательно входит в минимальное остовное дерево.

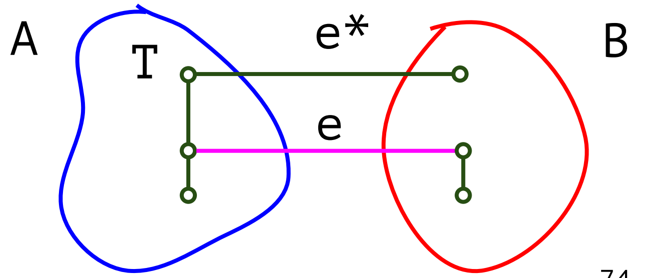
Доказательство: предположим, что это не так => e не входит в минимальное остовное дерево (обозначим его **T**). Тогда существует e* ∈ T, пересекающее (A, B). Заметим, что |e*| > |e|.

Далее попробуем взять T* = T - {e*} + {e}. Но всегда ли T* - это минимальное остовное дерево? Нет! Замена e* на e может создать цикл!

Но есть и хорошая новость: рассмотрим ребро f. Что можно о нем сказать?

T** = T - {f} + {e} - это МОД.

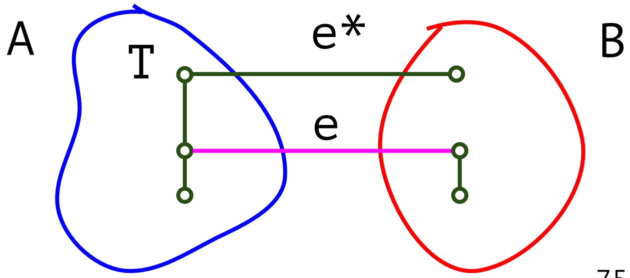
А связный он т.к. из любой вершины A все еще есть путь в B. Любые пути, включавшие f, теперь включают e.



Лемма #4 (свойство разреза): пусть есть некоторый разрез (A, B). Если ребро e - кратчайшее из пересекающих этот разрез ребер, то оно обязательно входит в минимальное остовное дерево.

Доказательство: предположим, что это не так => e не входит в минимальное остовное дерево (обозначим его **T**).

Тогда либо существует $e^* \in T$, пересекающее (A, B), такое что замена e* на e не создаст циклов, т.е. T* = T - {e*} + e - МОД.

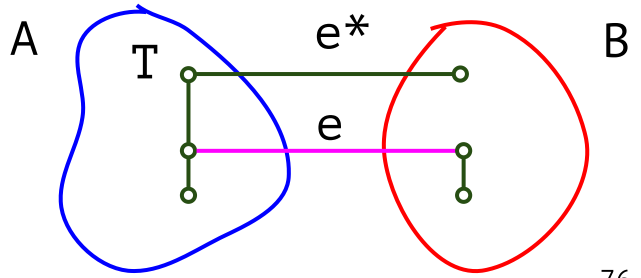


Лемма #4 (свойство разреза): пусть есть некоторый разрез (A, B). Если ребро e - кратчайшее из пересекающих этот разрез ребер, то оно обязательно входит в минимальное остовное дерево.

Доказательство: предположим, что это не так => e не входит в минимальное остовное дерево (обозначим его **T**).

Тогда либо существует $e^* \in T$, пересекающее (A, B), такое что замена e* на e не создаст циклов, т.е. $T^* = T - \{e^*\} + e$ - МОД.

Либо существует $f \in T$, пересекающее (A, B), f и e - два пересекающих ребра цикла в $T^{**} = T + \{e\}$. И тогда $T^{***} = T + \{e\} - \{f\}$ - это МОД.



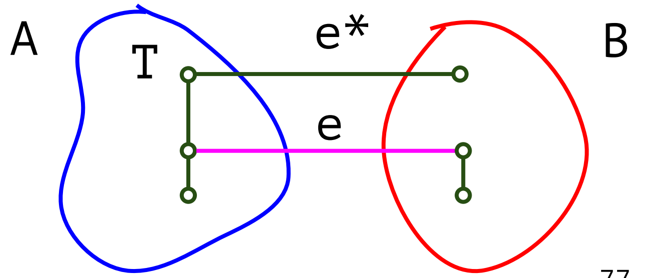
Лемма #4 (свойство разреза): пусть есть некоторый разрез (A, B). Если ребро e - кратчайшее из пересекающих этот разрез ребер, то оно обязательно входит в минимальное остовное дерево.

Доказательство: предположим, что это не так => e не входит в минимальное остовное дерево (обозначим его **T**).

Тогда либо существует $e^* \in T$, пересекающее (A, B), такое что замена e* на e не создаст циклов, т.е. $T^* = T - \{e^*\} + e$ - МОД.

Либо существует $f \in T$, пересекающее (A, B), f и e - два пересекающих ребра цикла в $T^{**} = T + \{e\}$. И тогда $T^{***} = T + \{e\} - \{f\}$ - это МОД.

В любом случае, суммарный вес ребер T* или T*** меньше суммарного веса ребер T.



Лемма #4 (свойство разреза): пусть есть некоторый разрез (A, B). Если ребро e - кратчайшее из пересекающих этот разрез ребер, то оно обязательно входит в минимальное остовное дерево.

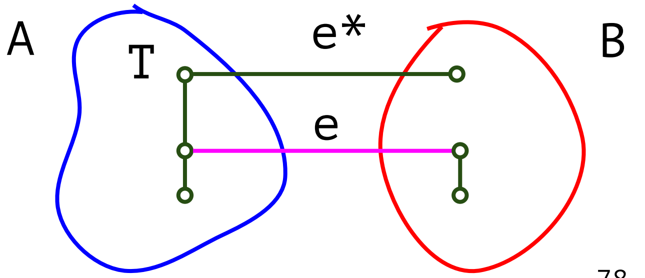
Доказательство: предположим, что это не так => e не входит в минимальное остовное дерево (обозначим его **T**).

Тогда либо существует $e^* \in T$, пересекающее (A, B), такое что замена e* на e не создаст циклов, т.е. $T^* = T - \{e^*\} + e$ - МОД.

Либо существует $f \in T$, пересекающее (A, B), f и e - два пересекающих ребра цикла в $T^{**} = T + \{e\}$. И тогда $T^{***} = T + \{e\} - \{f\}$ - это МОД.

В любом случае, суммарный вес ребер T* или T*** меньше суммарного веса ребер T.

Противоречие с предположением о том, что e не входит в МОД. □



Алгоритм Прима: корректность, часть #2

Утверждение: алгоритм Прима находит минимальное остовное дерево в графе.

Доказательство: уже показали, что получаем какое-то остовное дерево.

Лемма #4 (свойство разреза): пусть есть некоторый разрез (A, B). Если ребро e - кратчайшее из пересекающих этот разрез ребер, то оно входит в минимальное остовное дерево.

Алгоритм Прима: корректность, часть #2

Утверждение: алгоритм Прима находит минимальное остовное дерево в графе.

Доказательство: уже показали, что получаем какое-то остовное дерево. Но по построению алгоритма мы каждый раз берем кратчайшее ребро e пересекающее разрез (X, V\X) =>

Лемма #4 (свойство разреза): пусть есть некоторый разрез (A, B). Если ребро e - кратчайшее из пересекающих этот разрез ребер, то оно входит в минимальное остовное дерево.

Алгоритм Прима: корректность, часть #2

Утверждение: алгоритм Прима находит минимальное остовное дерево в графе.

Доказательство: уже показали, что получаем какое-то остовное дерево. Но по построению алгоритма мы каждый раз берем кратчайшее ребро e пересекающее разрез (X, V\X) => каждое его ребро из минимального остовного дерева =>

Лемма #4 (свойство разреза): пусть есть некоторый разрез (A, B). Если ребро e - кратчайшее из пересекающих этот разрез ребер, то оно входит в минимальное остовное дерево.

Алгоритм Прима: корректность, часть #2

Утверждение: алгоритм Прима находит минимальное остовное дерево в графе.

Доказательство: уже показали, что получаем какое-то остовное дерево. Но по построению алгоритма мы каждый раз берем кратчайшее ребро e пересекающее разрез (X, V\X) => каждое его ребро из минимального остовного дерева => оно всё является подмножеством минимального =>

Лемма #4 (свойство разреза): пусть есть некоторый разрез (A, B). Если ребро e - кратчайшее из пересекающих этот разрез ребер, то оно входит в минимальное остовное дерево.

Алгоритм Прима: корректность, часть #2

Утверждение: алгоритм Прима находит минимальное остовное дерево в графе.

Доказательство: уже показали, что получаем какое-то остовное дерево. Но по построению алгоритма мы каждый раз берем кратчайшее ребро e пересекающее разрез (X, V\X) => каждое его ребро из минимального остовного дерева => оно все является подмножеством минимального => оно и есть искомое минимальное остовное дерево.

Лемма #4 (свойство разреза): пусть есть некоторый разрез (A, B). Если ребро e - кратчайшее из пересекающих этот разрез ребер, то оно входит в минимальное остовное дерево.

Алгоритм Прима: сложность

Какова временная сложность алгоритма Прима?

Алгоритм Прима: сложность

Какова временная сложность алгоритма Прима?

При наивной реализации мы |V| раз будем искать минимальное ребро (перебором по |E| ребрам), т.е. получим O(|V|*|E|).

Алгоритм Прима: сложность

Какова временная сложность алгоритма Прима?

При наивной реализации мы |V| раз будем искать минимальное ребро (перебором по |E| ребрам), т.е. получим O(|V|*|E|).

Как улучшить?

Алгоритм Прима: сложность

Какова временная сложность алгоритма Прима?

При наивной реализации мы |V| раз будем искать минимальное ребро (перебором по |E| ребрам), т.е. получим O(|V|*|E|).

Как улучшить? Хипы!

Алгоритм Прима: сложность

Какова временная сложность алгоритма Прима?

Реализация через ~~хипы~~ очереди с приоритетом:

1) храним вершины (!) из V\X

Алгоритм Прима: сложность

Какова временная сложность алгоритма Прима?

Реализация через ~~хипы~~ очереди с приоритетом:

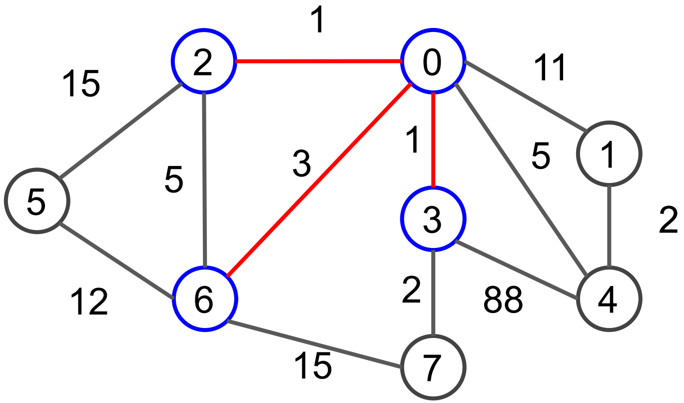
1) храним вершины (!) из V\X
2) в качестве значения (приоритета) используется вес минимального ребра из вершины в X.

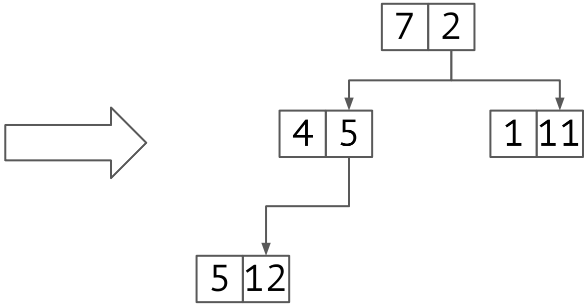
Алгоритм Прима: сложность

Какова временная сложность алгоритма Прима?

Реализация через ~~хипы~~ очереди с приоритетом:

1) храним вершины (!) из V\X
2) в качестве значения (приоритета) используется вес минимального ребра из вершины в X.



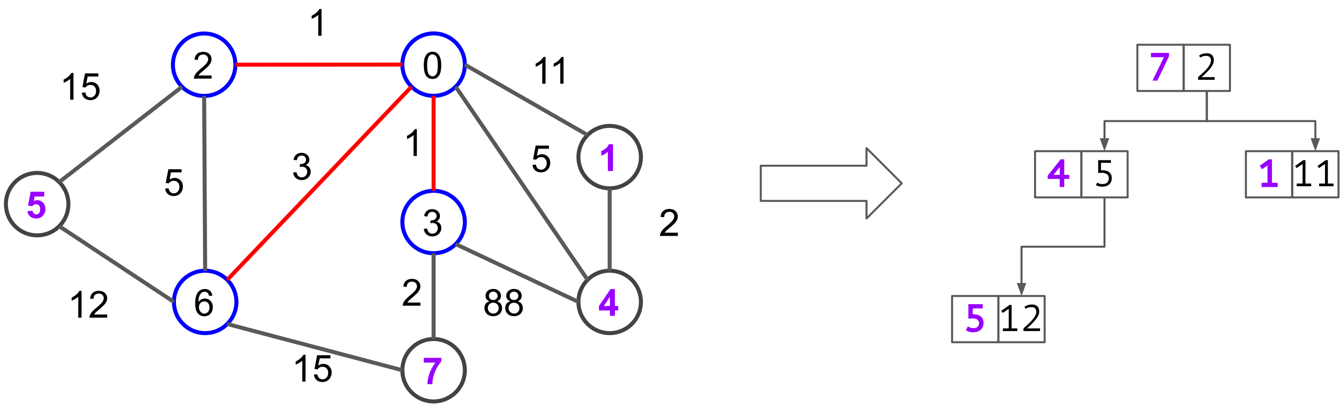


Алгоритм Прима: сложность

Какова временная сложность алгоритма Прима?

Реализация через ~~хипы~~ очереди с приоритетом:

1) храним вершины (!) из V\X
2) в качестве значения (приоритета) используется вес минимального ребра из вершины в X.

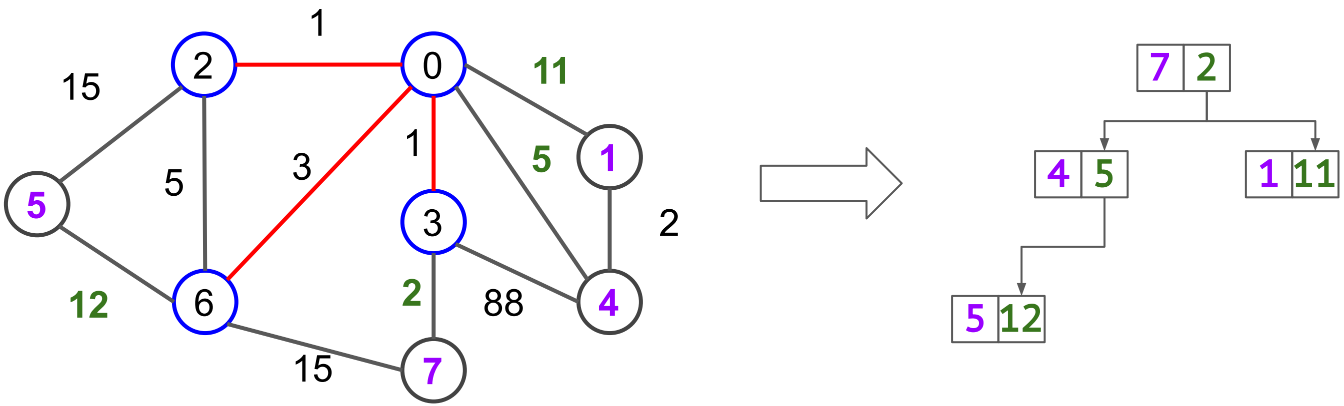


Алгоритм Прима: сложность

Какова временная сложность алгоритма Прима?

Реализация через ~~хипы~~ очереди с приоритетом:

1) храним вершины (!) из V\X
2) в качестве значения (приоритета) используется вес минимального ребра из вершины в X.



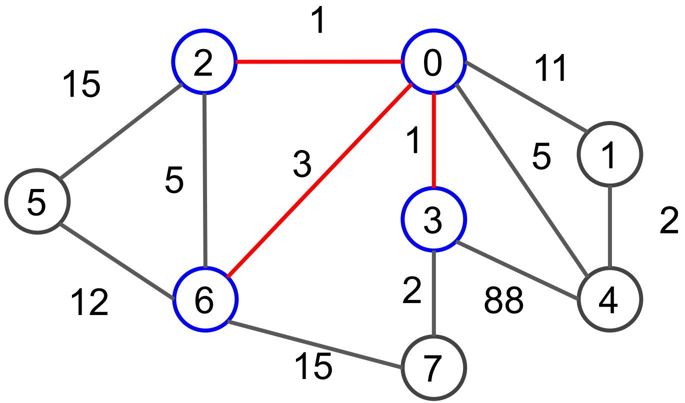
Алгоритм Прима: сложность

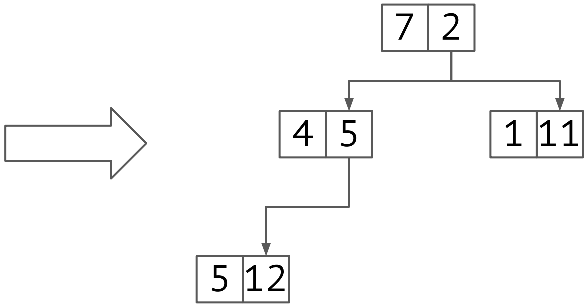
Какова временная сложность алгоритма Прима?

Реализация через ~~хипы~~ очереди с приоритетом:

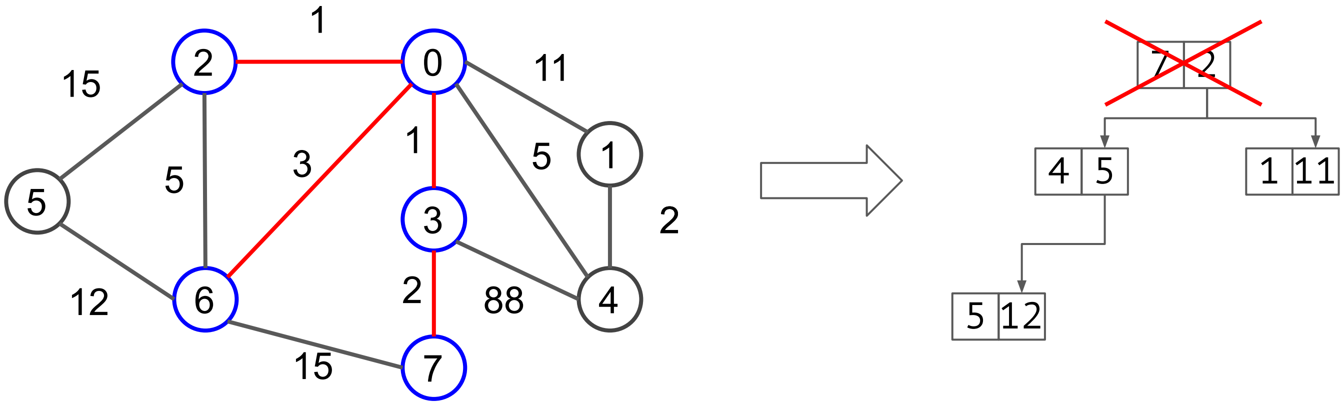
1) храним вершины (!) из V\X
2) в качестве значения (приоритета) используется вес минимального ребра из вершины в X.
3) на каждом шаге алгоритма достаем минимум из хипа
4) но при этом нужно обновить приоритет всем смежным вершинам!

Алгоритм Прима: сложность

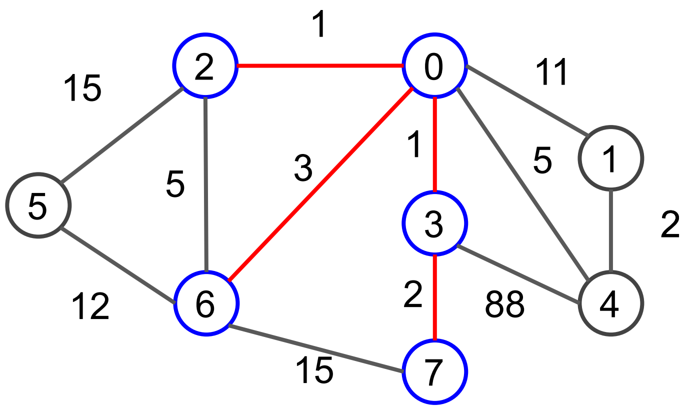


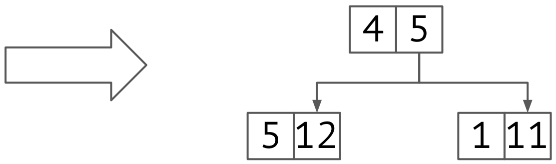


Алгоритм Прима: сложность

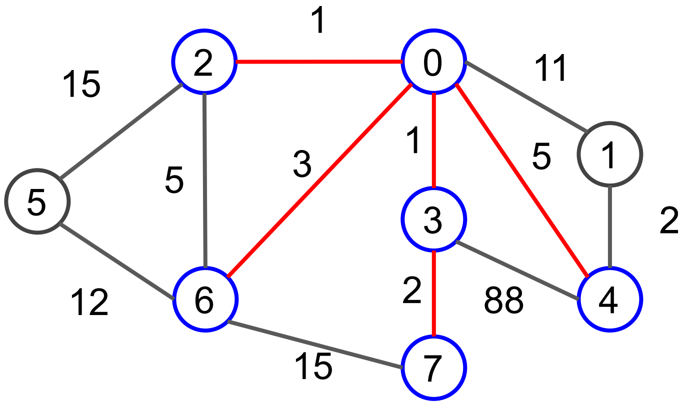


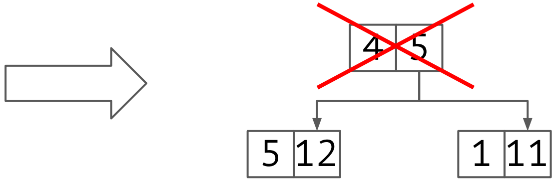
Алгоритм Прима: сложность





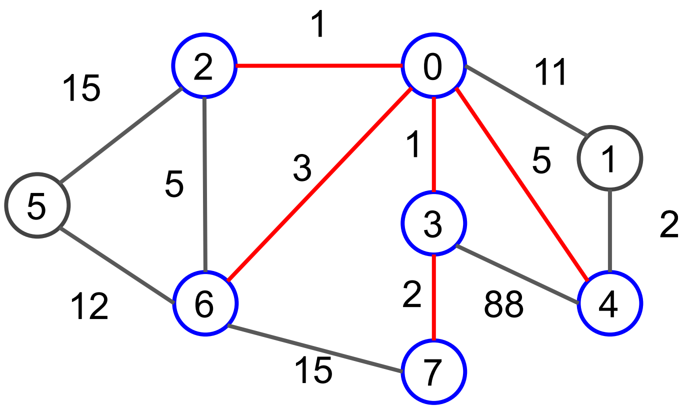
Алгоритм Прима: сложность

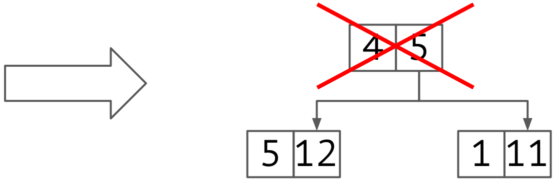




Но просто просеивание запустить недостаточно, еще нужно обновить 11!

Алгоритм Прима: сложность

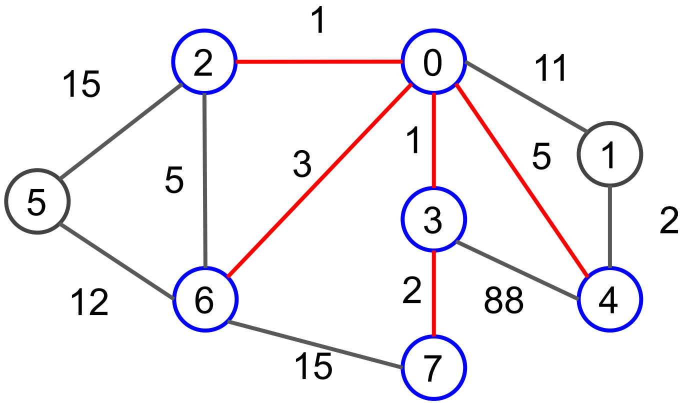


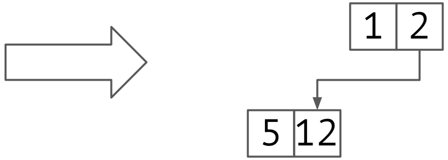


Но просто просеивание запустить недостаточно, еще нужно обновить 11!

Но знаем, на что обновить: на ребро из 4 (которую только что добавили)

Алгоритм Прима: сложность





Но просто просеивание запустить недостаточно, еще нужно обновить 11!

Но знаем, на что обновить: на ребро из 4 (которую только что добавили)

Алгоритм Прима: сложность

Алгоритм Прима через хипы:

0) Инициализируем хип: добавляем все вершины (значение для каждой - это вес минимального исходящего из нее ребра)

Алгоритм Прима: сложность

Алгоритм Прима через хипы:

0) Инициализируем хип: добавляем все вершины (значение для каждой - это вес минимального исходящего из нее ребра)

При правильном хранении графа займет $O(|E| + |V|log(|V|))$

Алгоритм Прима: сложность

Алгоритм Прима через хипы:

0) Инициализируем хип: добавляем все вершины (значение для каждой - это вес минимального исходящего из нее ребра)

При правильном хранении графа займет $O(|E| + |V|log(|V|))$

переборы для поиска минимальных весов ребер

добавления вершин в хип

Алгоритм Прима: сложность

Алгоритм Прима через хипы:

0) Инициализируем хип: добавляем все вершины (значение для каждой - это вес минимального исходящего из нее ребра)

При правильном хранении графа займет $O(|E| + |V|)$

переборы для поиска минимальных весов ребер

добавления вершин в хип

*На самом деле построение кучи делается за линию

Алгоритм Прима: сложность

Алгоритм Прима через хипы:

0) Инициализируем хип: добавляем все вершины (значение для каждой - это вес минимального исходящего из нее ребра)

При правильном хранении графа займет $O(|E| + |V|)$

Но в связном графе: $|V| \leq |E|$ + 1,

Поэтому получаем: $O(|E|)$

переборы для поиска минимальных весов ребер

добавления вершин в хип

Алгоритм Прима: сложность

Алгоритм Прима через хипы:

0) Инициализируем хип: добавляем все вершины (значение для каждой - это вес минимального исходящего из нее ребра)

1) На каждой итерации делаем:

   a) extract_min из хипа за $O(log(|V|))$

Алгоритм Прима: сложность

Алгоритм Прима через хипы:

0) Инициализируем хип: добавляем все вершины (значение для каждой - это вес минимального исходящего из нее ребра)

1) На каждой итерации делаем:

   a) extract_min из хипа за $O(log(|V|))$
   b) обновление приоритета смежным с полученной вершинам (каждая операция делается за $O(log(|V|))$)

Алгоритм Прима: сложность

Алгоритм Прима через хипы:

0) Инициализируем хип: добавляем все вершины (значение для каждой - это вес минимального исходящего из нее ребра)

1) На каждой итерации делаем:

   a) extract_min из хипа за $O(log(|V|))$
   b) обновление приоритета смежным с полученной вершинам (каждая операция делается за $O(log(|V|))$)

   Всего таких операций будет не больше, чем ребер в графе!

Алгоритм Прима: сложность

Какова временная сложность алгоритма Прима?

При наивной реализации мы |V| раз будем искать минимальное ребро (перебором по |E| ребрам), т.е. получим O(|V|*|E|).

При реализации через бинарные хипы: $O(|V| * log(|V|) + |E| * log(|V|))$

Алгоритм Прима: сложность

Какова временная сложность алгоритма Прима?

При наивной реализации мы |V| раз будем искать минимальное ребро (перебором по |E| ребрам), т.е. получим O(|V|*|E|).

При реализации через бинарные хипы: $O(|V| * log(|V|) + |E| * log(|V|))$

Но т.к. граф связный, то получаем: $O(|E| * log(|V|))$

Алгоритм Прима: сложность

Какова временная сложность алгоритма Прима?

При наивной реализации мы |V| раз будем искать минимальное ребро (перебором по |E| ребрам), т.е. получим O(|V|*|E|).

При реализации через бинарные хипы: $O(|V| * log(|V|) + |E| * log(|V|))$

Но т.к. граф связный, то получаем: $O(|E| * log(|V|))$

Вопрос: а через фибоначчиевы пирамиды?

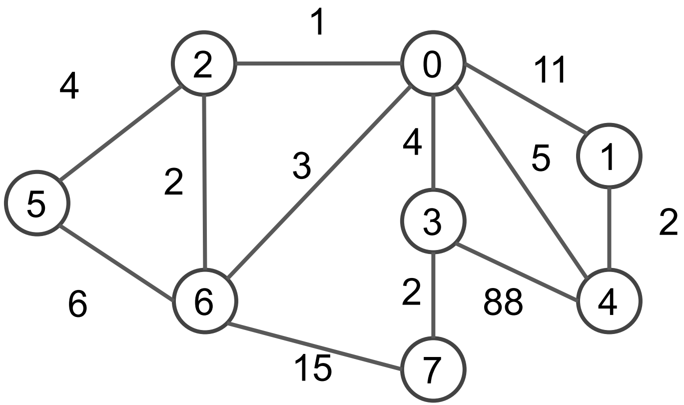
Мини-задача #34 (2 балла)

Решаем задачу на поиск минимального остовного дерева:

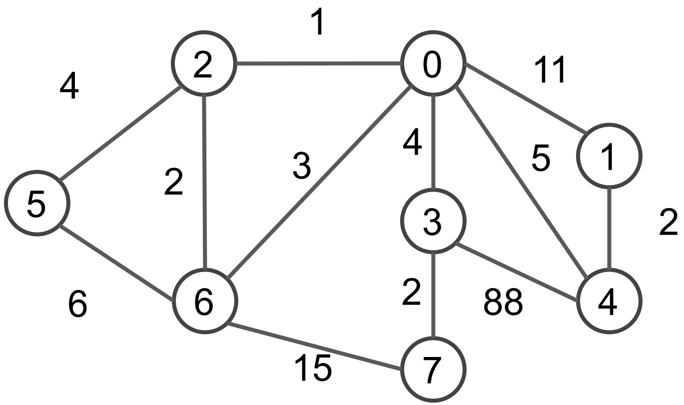
https://classroom.github.com/assignment-invitations/b4a7
21959e995116b46c1274103134a9/status

Используйте для этого алгоритм Прима с оптимизацией через хипы.

Алгоритм Краскала

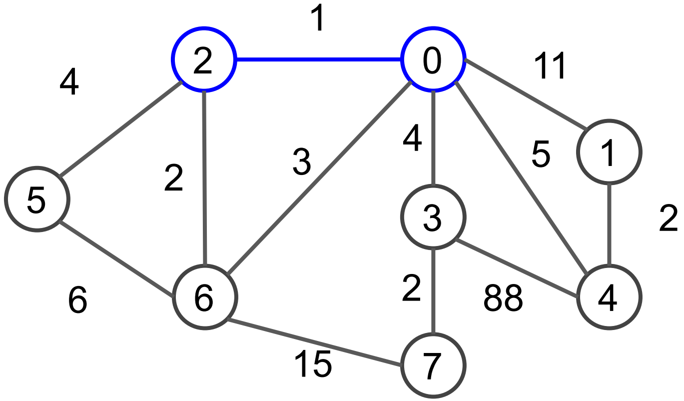


Алгоритм Краскала



0) Сортируем все ребра в порядке возрастания веса

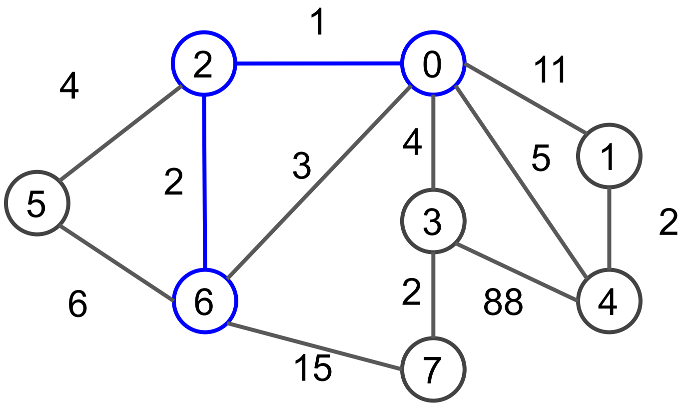
Алгоритм Краскала



0) Сортируем все ребра в порядке возрастания веса

1) Идем по получившемуся массиву ребер и добавляем ребро в результат T, …

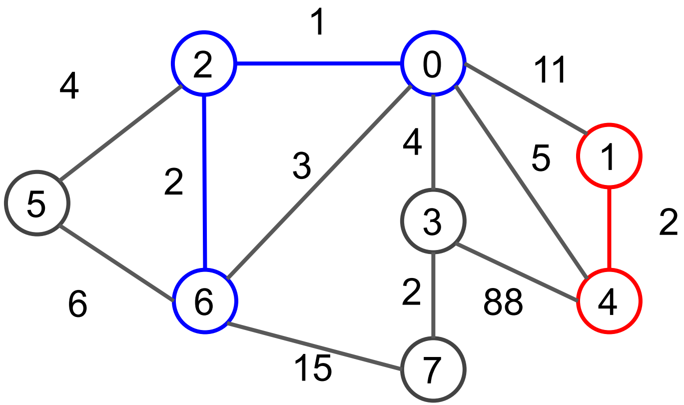
Алгоритм Краскала



0) Сортируем все ребра в порядке возрастания веса

1) Идем по получившемуся массиву ребер и добавляем ребро в результат T, но только, если это не создаст циклов!

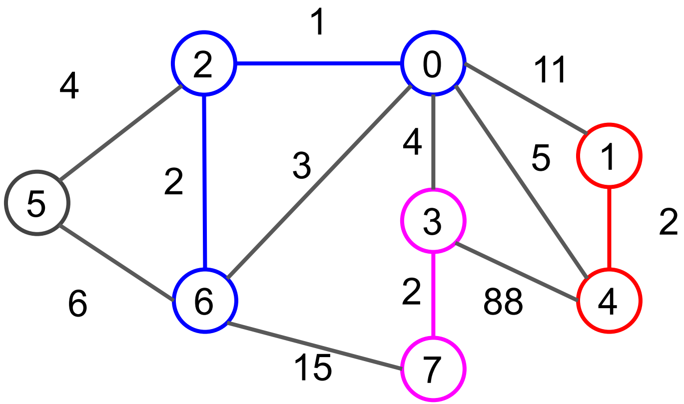
Алгоритм Краскала



0) Сортируем все ребра в порядке возрастания веса

1) Идем по получившемуся массиву ребер и добавляем ребро в результат T, но только, если это не создаст циклов!

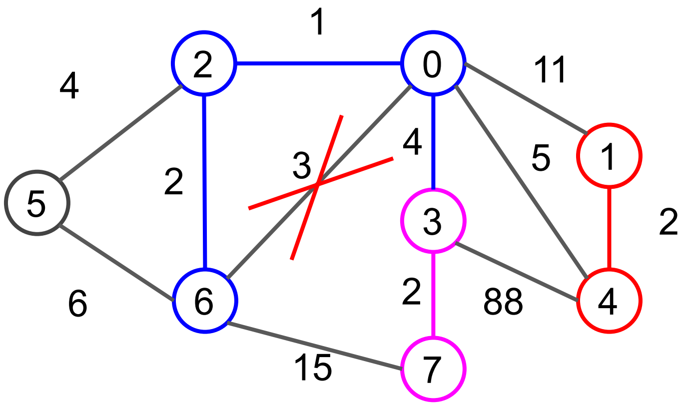
Алгоритм Краскала



0) Сортируем все ребра в порядке возрастания веса

1) Идем по получившемуся массиву ребер и добавляем ребро в результат T, но только, если это не создаст циклов!

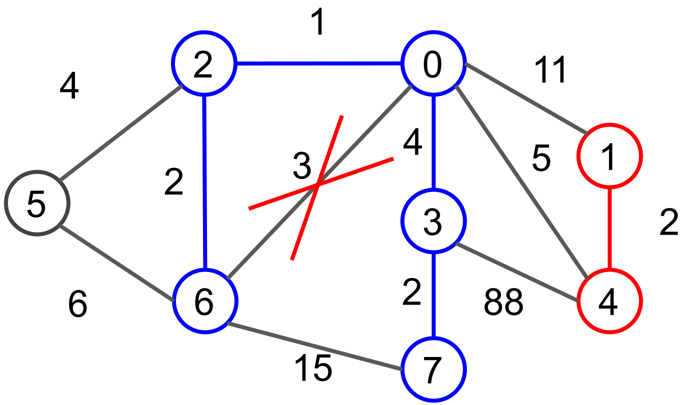
Алгоритм Краскала



0) Сортируем все ребра в порядке возрастания веса

1) Идем по получившемуся массиву ребер и добавляем ребро в результат Т, но только, если это не создаст циклов!

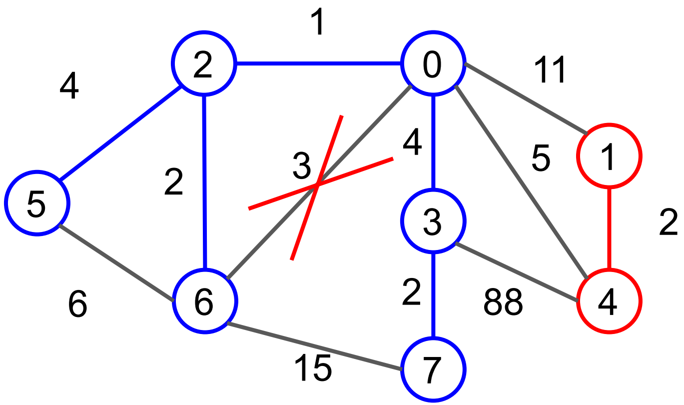
Алгоритм Краскала



0) Сортируем все ребра в порядке возрастания веса

1) Идем по получившемуся массиву ребер и добавляем ребро в результат T, но только, если это не создаст циклов!

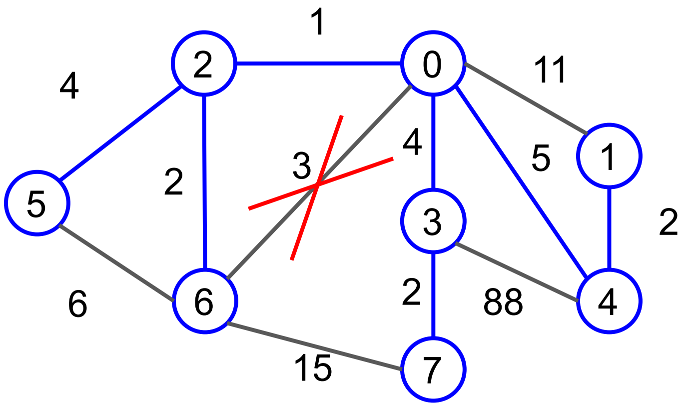
Алгоритм Краскала



0) Сортируем все ребра в порядке возрастания веса

1) Идем по получившемуся массиву ребер и добавляем ребро в результат T, но только, если это не создаст циклов!

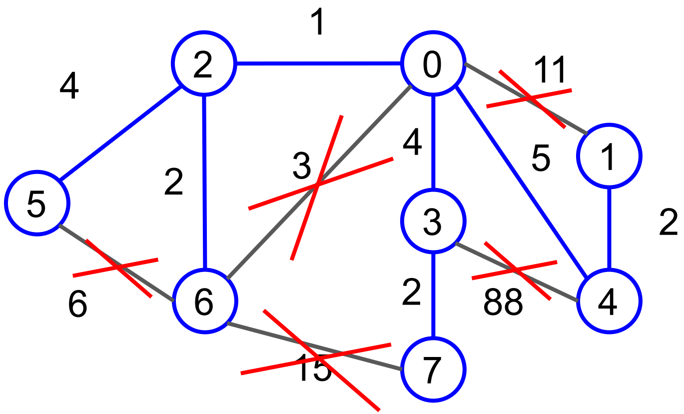
Алгоритм Краскала



0) Сортируем все ребра в порядке возрастания веса

1) Идем по получившемуся массиву ребер и добавляем ребро в результат T, но только, если это не создаст циклов!

Алгоритм Краскала

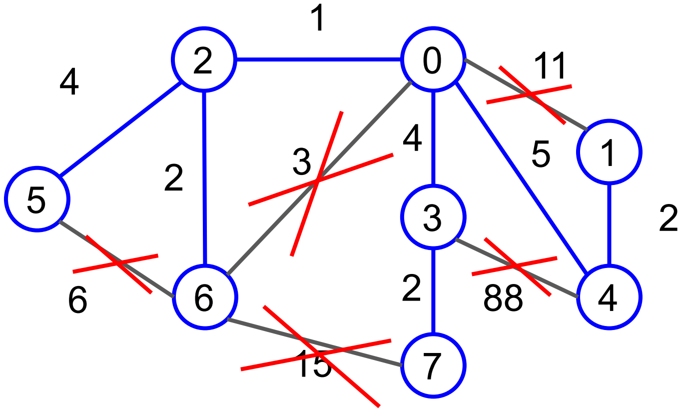


0) Сортируем все ребра в порядке возрастания веса

1) Идем по получившемуся массиву ребер и добавляем ребро в результат T, но только, если это не создаст циклов!

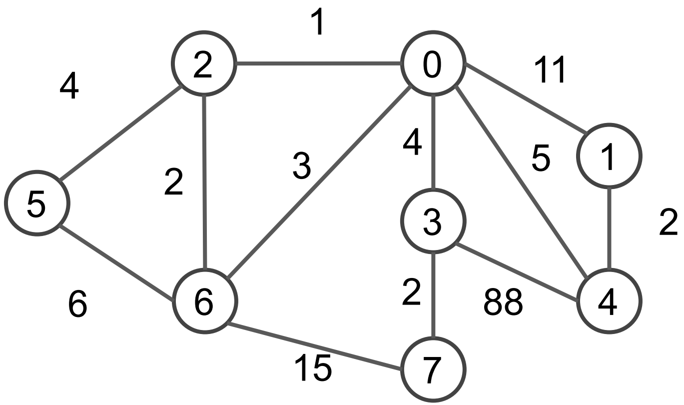
2) Полученное T - ребра для искомого остовного дерева.

Алгоритм Краскала: корректность



0) По построению у нас нет циклов (старались на каждой итерации, чтобы этого избежать).

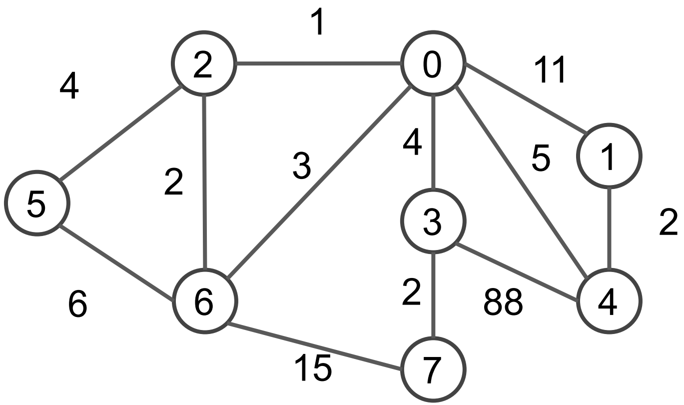
Алгоритм Краскала: корректность



0) По построению у нас нет циклов (старались на каждой итерации, чтобы этого избежать).

1) Почему он связный?

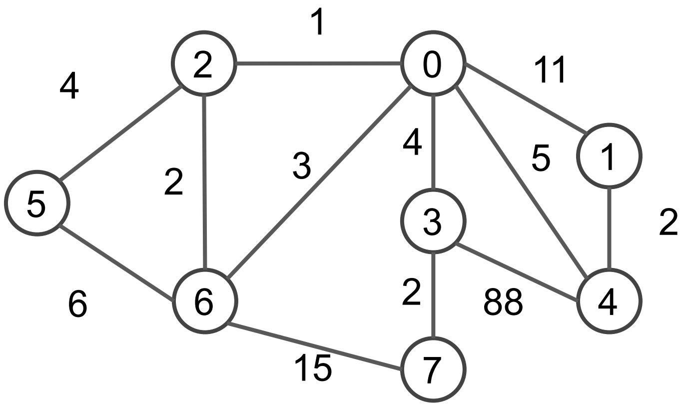
Алгоритм Краскала: корректность



0) По построению у нас нет циклов (старались на каждой итерации, чтобы этого избежать).

1) Почему он связный? Покажем, что какой бы мы не взяли разрез, в T будет ребро, которое его пересекает.

Алгоритм Краскала: корректность



0) По построению у нас нет циклов (старались на каждой итерации, чтобы этого избежать).

1) Почему он связный? Покажем, что какой бы мы не взяли разрез, в T будет ребро, которое его пересекает.

А тогда по лемме 1 граф будет связным

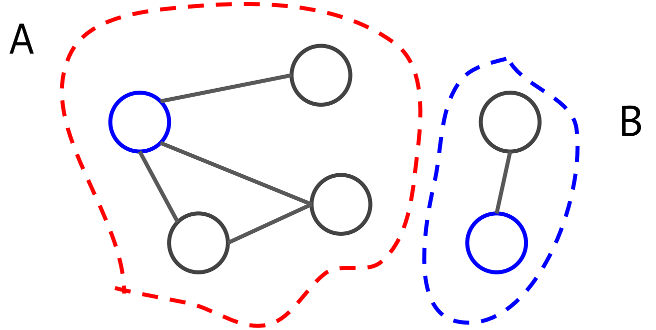
Тривиальные свойства разрезов

Лемма #1: граф несвязный ⇔ ∃ разрез (A, B) без пересекающих ребер

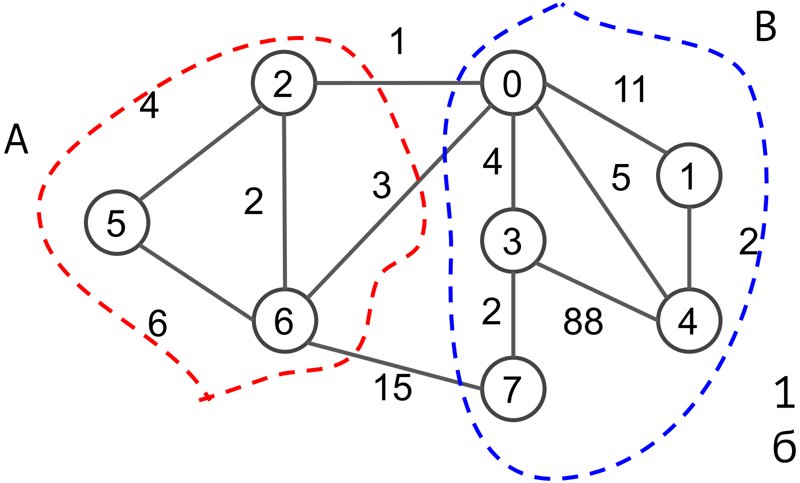
Доказательство:

⇒ граф несвязный, значит есть хотя бы две компоненты связности, возьмем одну из них в качестве A, все остальное в качестве B. Получили искомый разрез.

⇐ возьмем одну вершину из A, вторую из B; по условию не может быть пути из одной из этих вершин, в другую, т.к. нет пересекающих ребер



Алгоритм Краскала: корректность

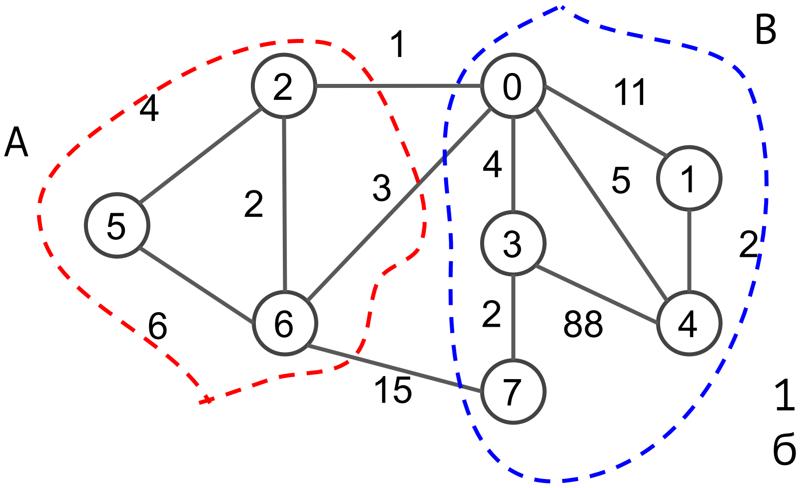


0) По построению у нас нет циклов (старались на каждой итерации, чтобы этого избежать).

1) Почему он связный? Покажем, что какой бы мы не взяли разрез, в T будет ребро, которое его пересекает.

Зафиксируем разрез (A, B).

Алгоритм Краскала: корректность

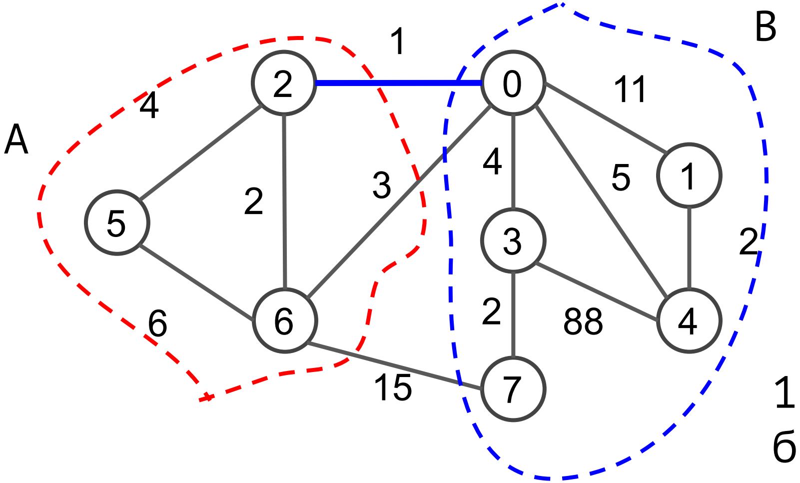


0) По построению у нас нет циклов (старались на каждой итерации, чтобы этого избежать).

1) Почему он связный? Покажем, что какой бы мы не взяли разрез, в T будет ребро, которое его пересекает.

Зафиксируем разрез (A, B). Алгоритм рассматривает все ребра => все пересекающие разрез ребра он тоже рассмотрит.
Какое ребро он рассмотрит первым?

Алгоритм Краскала: корректность

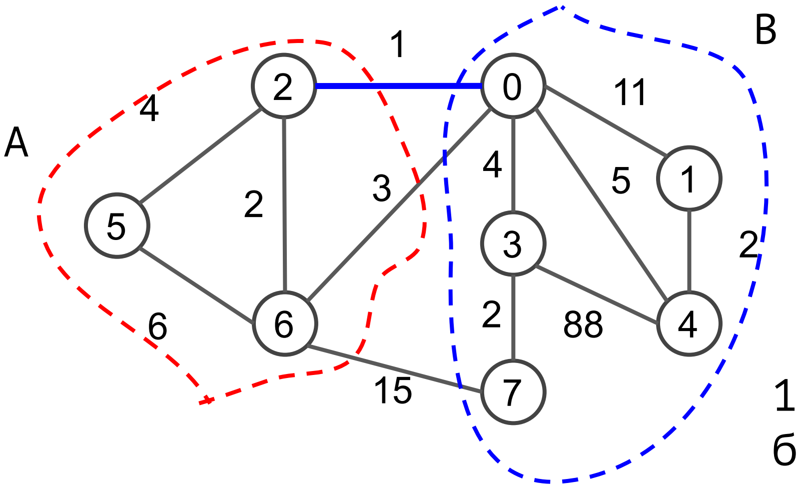


0) По построению у нас нет циклов (старались на каждой итерации, чтобы этого избежать).

1) Почему он связный? Покажем, что какой бы мы не взяли разрез, в T будет ребро, которое его пересекает.

Зафиксируем разрез (A, B). Алгоритм рассматривает все ребра => все пересекающие разрез ребра он тоже рассмотрит.
Какое ребро он рассмотрит первым?

Алгоритм Краскала: корректность

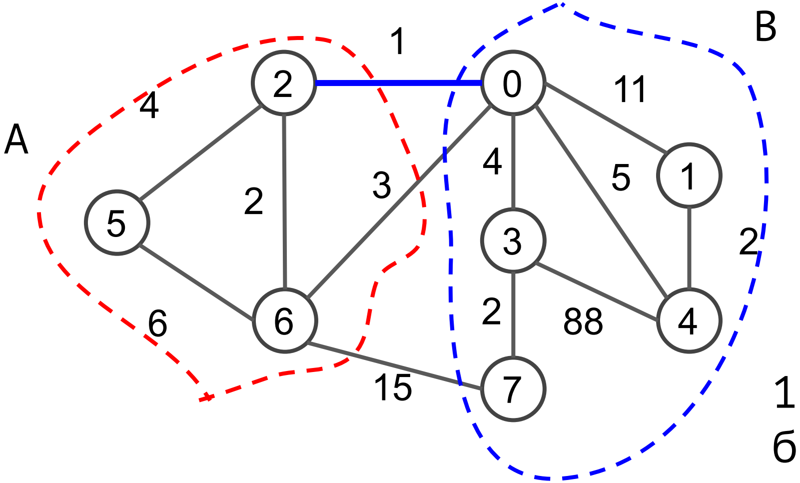


0) По построению у нас нет циклов (старались на каждой итерации, чтобы этого избежать).

1) Почему он связный? Покажем, что какой бы мы не взяли разрез, в T будет ребро, которое его пересекает.

Зафиксируем разрез (A, B). Алгоритм рассматривает все ребра => все пересекающие разрез ребра он тоже рассмотрит.
Какое ребро он рассмотрит первым? А добавит ли он это ребро в ответ?

Алгоритм Краскала: корректность

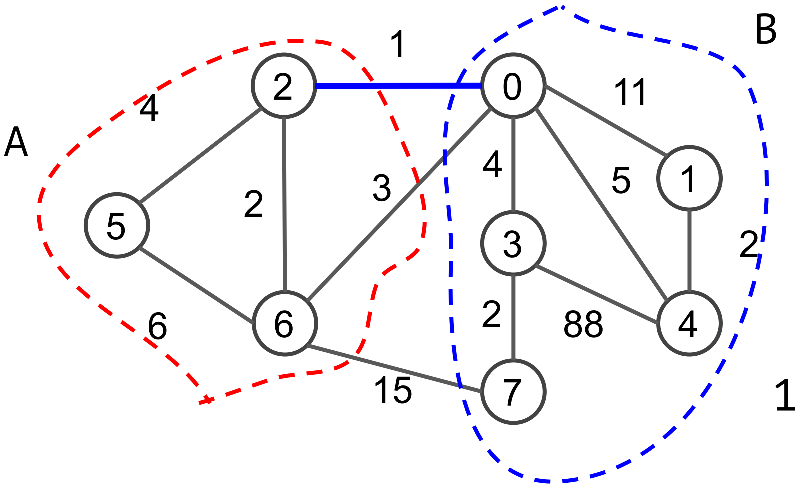


0) По построению у нас нет циклов (старались на каждой итерации, чтобы этого избежать).

1) Почему он связный? Покажем, что какой бы мы не взяли разрез, в T будет ребро, которое его пересекает.

Зафиксируем разрез (A, B). Алгоритм рассматривает все ребра => все пересекающие разрез ребра он тоже рассмотрит.
Какое ребро он рассмотрит первым? А добавит ли он это ребро в ответ? Да! По лемме #3 оно не образует цикл, ведь в T пока нет других пересекающих ребер.

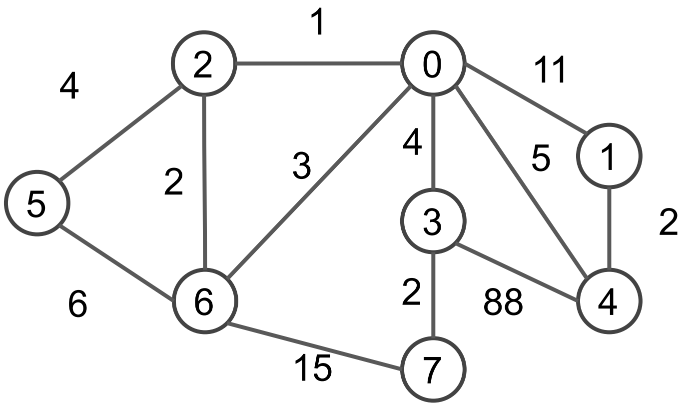
Алгоритм Краскала: корректность



0) По построению у нас нет циклов (старались на каждой итерации, чтобы этого избежать).

1) По лемме #3 граф будет связный.

Алгоритм Краскала: корректность

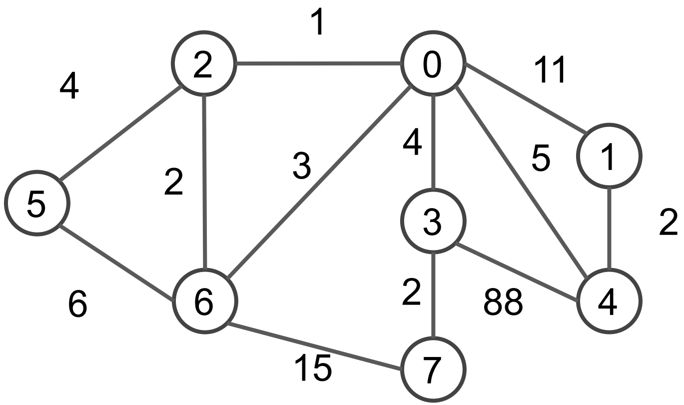


0) По построению у нас нет циклов (старались на каждой итерации, чтобы этого избежать).

1) По лемме #3 граф будет связный.

2) Почему полученное остовное дерево минимальное?

Алгоритм Краскала: корректность



0) По построению у нас нет циклов (старались на каждой итерации, чтобы этого избежать).

1) По лемме #3 граф будет связный.

2) Почему полученное остовное дерево минимальное? Потому, что на каждом шаге (когда ребро добавляется в ответ) выполняется свойство разреза (из леммы #4)

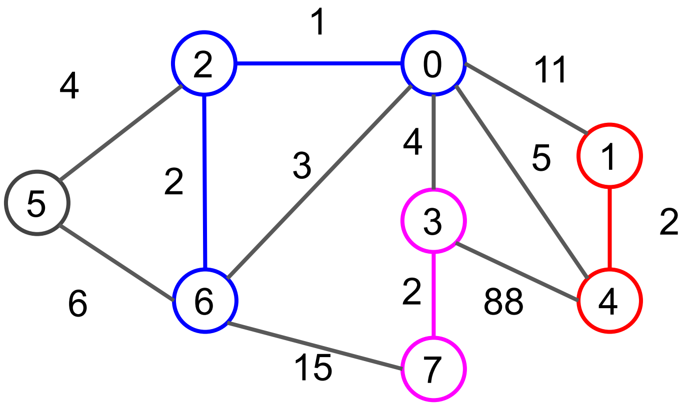
Алгоритм Прима: корректность, часть #2

Утверждение: алгоритм Прима находит минимальное остовное дерево в графе.

Лемма #4 (свойство разреза): пусть есть некоторый разрез (A, B). Если ребро e - кратчайшее из пересекающих этот разрез ребер, то оно обязательно входит в минимальное остовное дерево.

Доказательство: в курсе ДМТА, стр. 43 в методичке.

Алгоритм Краскала: корректность

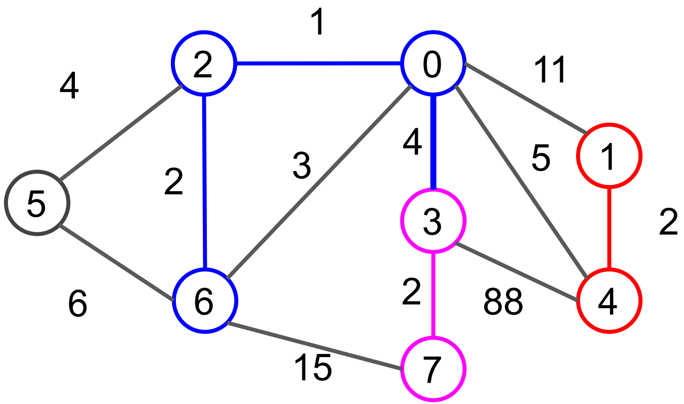


0) По построению у нас нет циклов (старались на каждой итерации, чтобы этого избежать).

1) По лемме #3 граф будет связный.

2) Почему полученное остовное дерево минимальное? Потому, что на каждом шаге (когда ребро добавляется в ответ) выполняется свойство разреза (из леммы #4)

Алгоритм Краскала: корректность

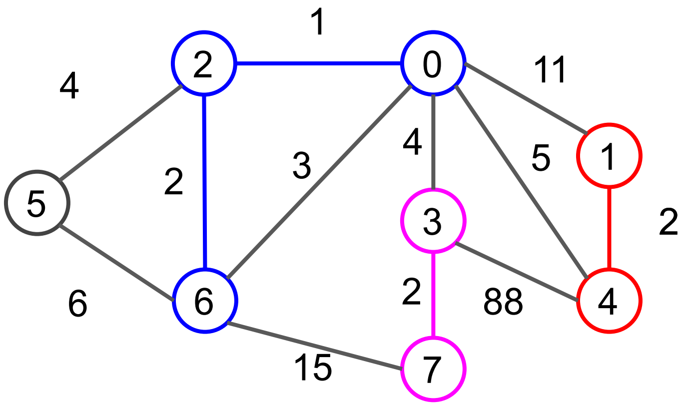


0) По построению у нас нет циклов (старались на каждой итерации, чтобы этого избежать).

1) По лемме #3 граф будет связный.

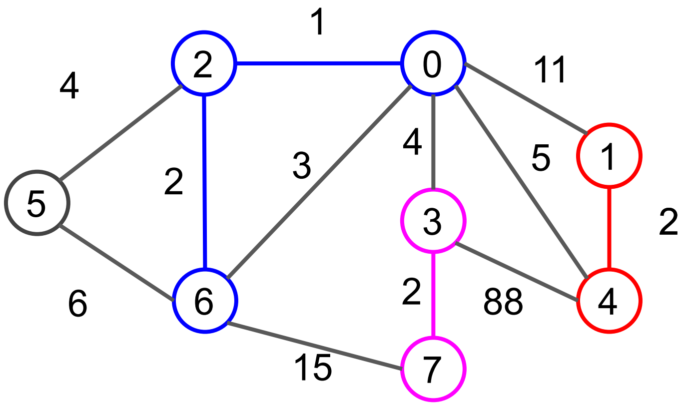
2) Почему полученное остовное дерево минимальное? Потому, что на каждом шаге (когда ребро добавляется в ответ) выполняется свойство разреза (из леммы #4)

Алгоритм Краскала: сложность



Сложность будет зависеть от реализации (как и всегда)

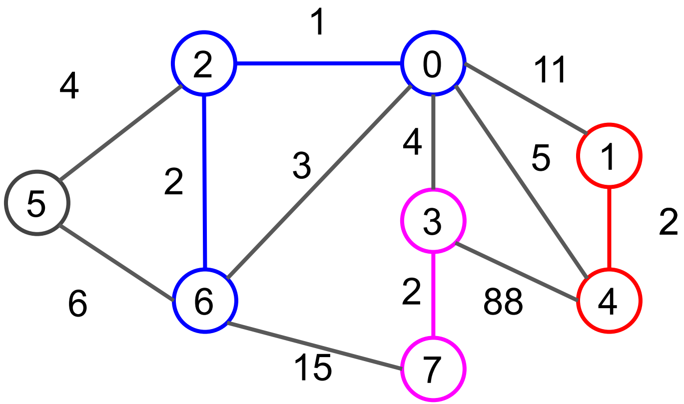
Алгоритм Краскала: сложность



Сложность будет зависеть от реализации

Наивная реализация: просто каждый раз проверяем на отсутствие циклов. Сложность?

Алгоритм Краскала: сложность

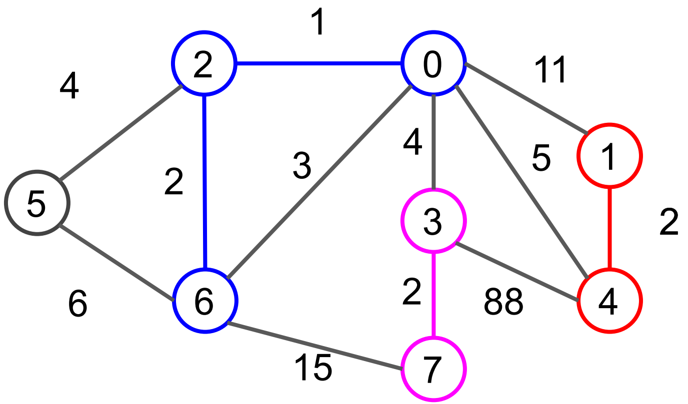


Сложность будет зависеть от реализации

Наивная реализация: просто каждый раз проверяем на отсутствие циклов. Сложность?

На каждой из |E| итераций поиск цикла за |V|, что дает O(|V|*|E|).

Алгоритм Краскала: сложность



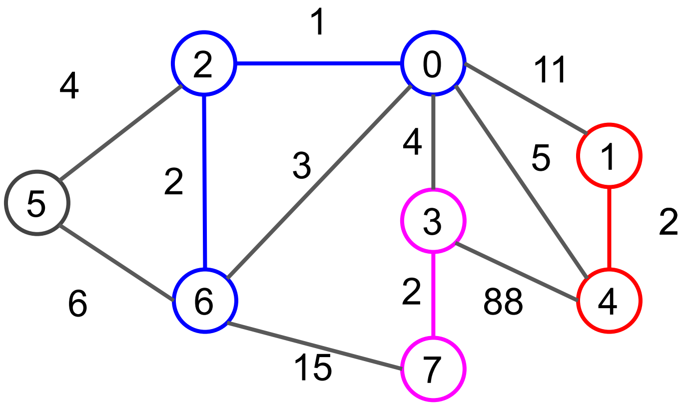
Сложность будет зависеть от реализации

Наивная реализация: просто каждый раз проверяем на отсутствие циклов. Сложность?

На каждой из |E| итераций поиск цикла за |V|, что дает O(|V|*|E|).

Не забываем про сортировку ребер за O(|E|log(|E|))

Алгоритм Краскала: сложность



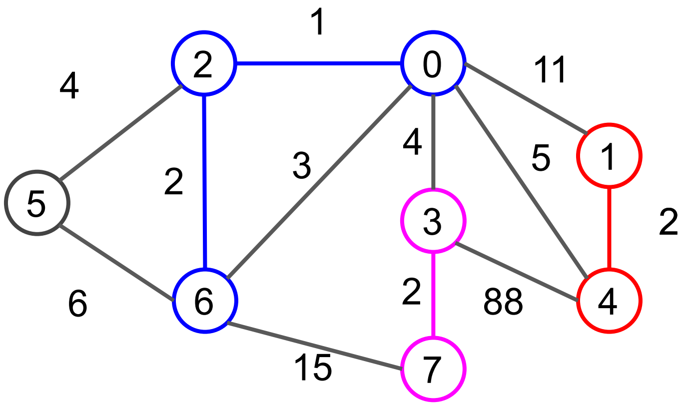
Сложность будет зависеть от реализации

Наивная реализация: просто каждый раз проверяем на отсутствие циклов. Сложность?

На каждой из |E| итераций поиск цикла за |V|, что дает O(|V|*|E|).

Не забываем про сортировку ребер за O(|E|log(|E|)), уточняем до O(|E|log(|V|)) из-за $|E| \leq |V|^2$.

Алгоритм Краскала: сложность



Сложность будет зависеть от реализации

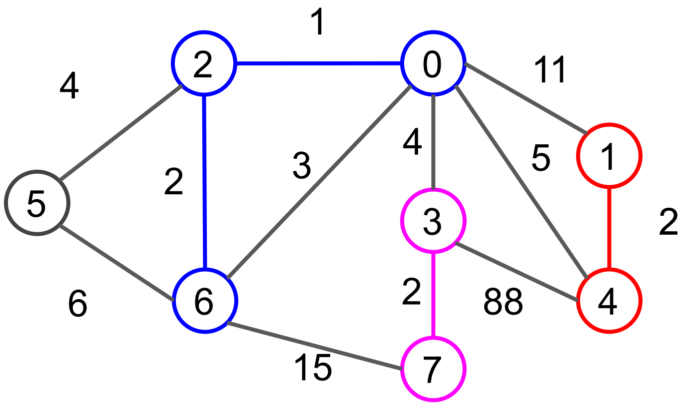
Наивная реализация: просто каждый раз проверяем на отсутствие циклов. Сложность?

На каждой из |E| итераций поиск цикла за |V|, что дает $\boxed{O(|V|*|E|)}$.

Не забываем про сортировку ребер за O(|E|log(|E|)), уточняем до O(|E|log(|V|)) из-за $|E| \leq |V|^2$.

Лучше, чем брутфорс, но все равно плохо (хуже Примы через хипы)

Алгоритм Краскала: union-find



На каждом шаге алгоритма имеем систему непересекающихся множеств (уже размеченных деревьев)

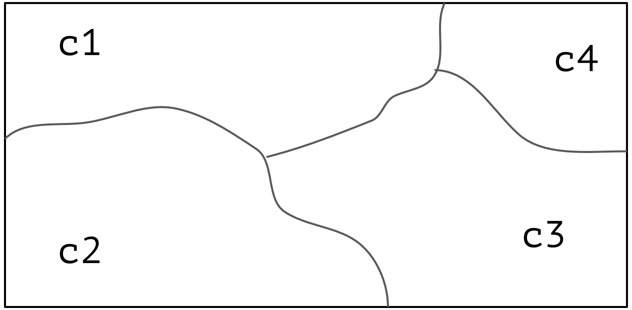
Для хранения таких множеств будем использовать специальную структуру - union-find.

Union-find

Union-find (он же disjoint-set, он же структура данных для непересекающихся множеств)

Union-find

Union-find (он же disjoint-set, он же структура данных для непересекающихся множеств)



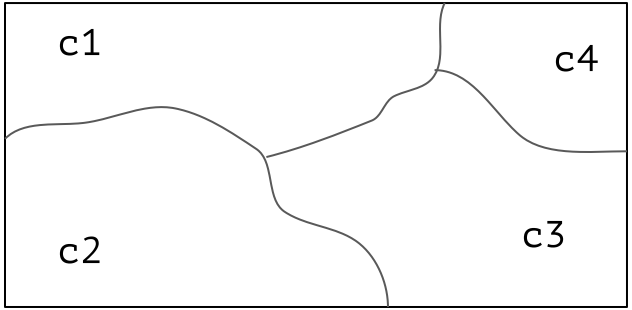
разбиение на классы эквивалентности

Union-find

Union-find (он же disjoint-set, он же структура данных для непересекающихся множеств)

Операции:

find(x) -> возвращает класс эквивалентности, к которому относится элемент x



разбиение на классы эквивалентности

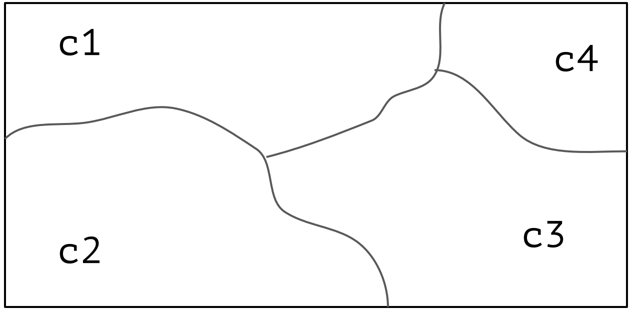
Union-find

Union-find (он же disjoint-set, он же структура данных для непересекающихся множеств)

Операции:

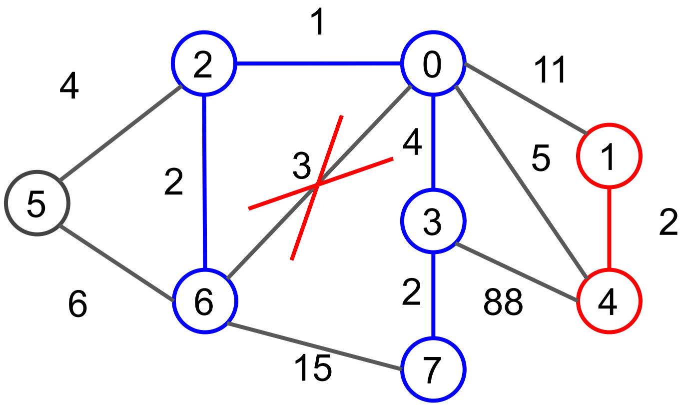
find(x) -> возвращает класс эквивалентности, к которому относится элемент x

union(c1, c2) -> объединяет два класса эквивалентности



разбиение на классы эквивалентности

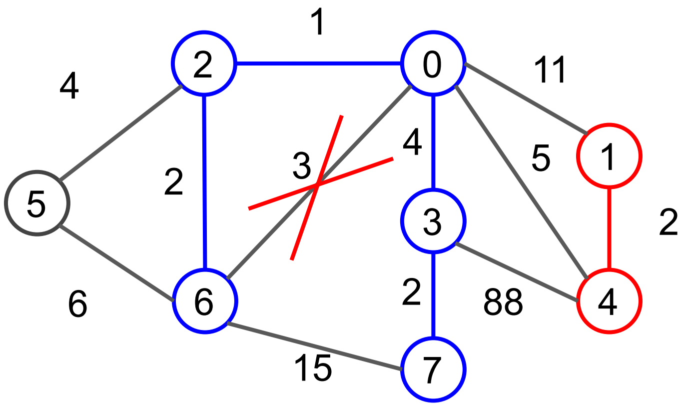
Алгоритм Краскала: union-find



0) Сортируем все ребра в порядке возрастания веса

1) Идем по получившемуся массиву ребер и добавляем ребро в результат T, но только, если это не создаст циклов!

Алгоритм Краскала: union-find

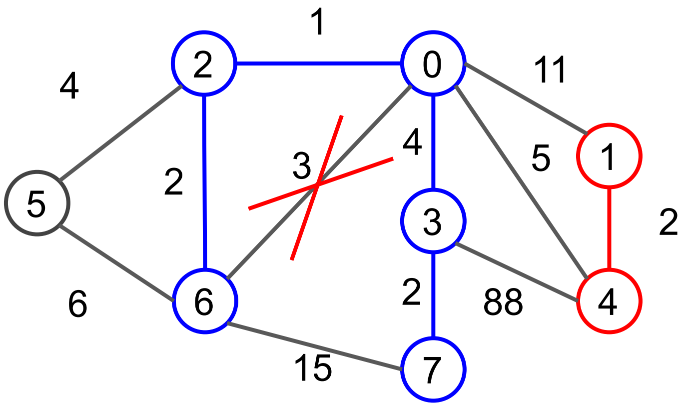


0) Сортируем все ребра в порядке возрастания веса

1) Храним все вершины в union-find

2) Идем по получившемуся массиву ребер и добавляем ребро в результат T, но только, вершины в разных классах эквивалентности!

Алгоритм Краскала: union-find



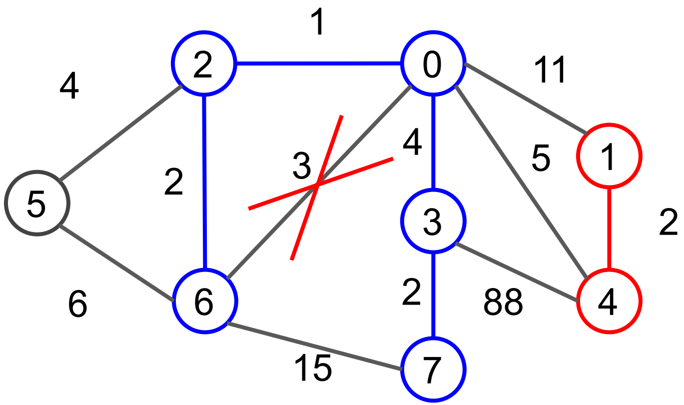
0) Сортируем все ребра в порядке возрастания веса

1) Храним все вершины в union-find

2) Идем по получившемуся массиву ребер и добавляем ребро в результат T, но только, вершины в разных классах эквивалентности!

3) При добавлении ребра сливаем классы экв. вершин

Алгоритм Краскала: union-find



0) Сортируем все ребра в порядке возрастания веса

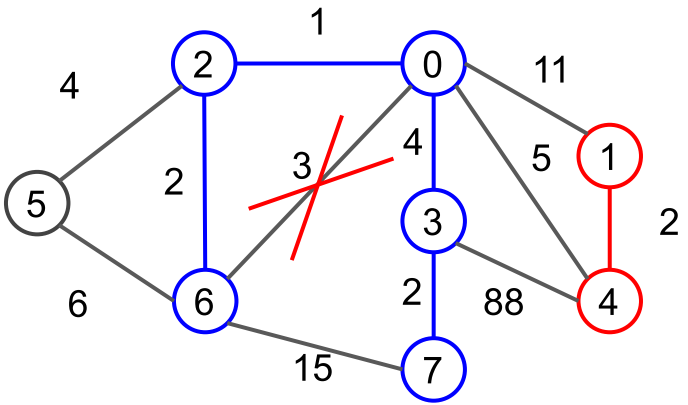
1) Храним все вершины в union-find

2) Идем по получившемуся массиву ребер и добавляем ребро в результат T, но только, вершины в разных классах эквивалентности!

3) При добавлении ребра сливаем классы экв. вершин

берем ребро (u,v), только если find(u) != find(v)

Алгоритм Краскала: union-find



0) Сортируем все ребра в порядке возрастания веса

1) Храним все вершины в union-find

2) Идем по получившемуся массиву ребер и добавляем ребро в результат T, но только, вершины в разных классах эквивалентности!

берем ребро (u,v), только если find(u) != find(v)

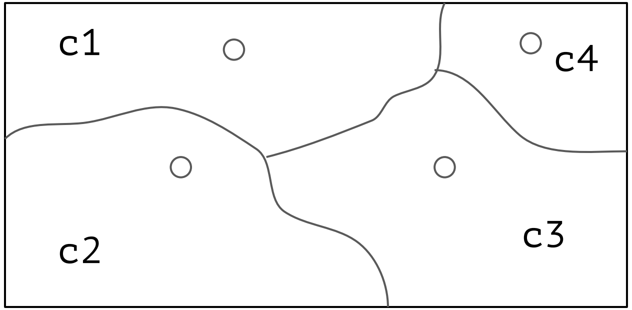
3) При добавлении ребра сливаем классы экв. вершин

после этого делаем union(find(u), find(v))

Union-find: реализация

Union-find (он же disjoint-set, он же структура данных для непересекающихся множеств)

Реализация (тривиальная):



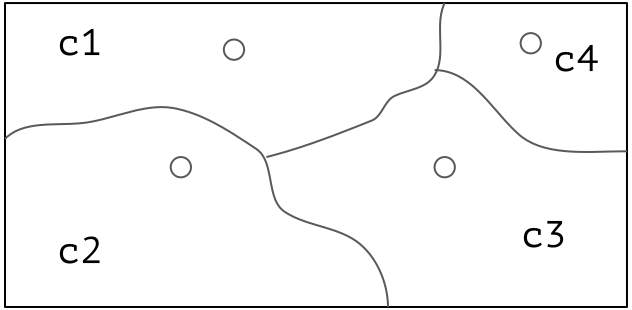
разбиение на классы эквивалентности

Union-find: реализация

Union-find (он же disjoint-set, он же структура данных для непересекающихся множеств)

Реализация (тривиальная):

1) Выбираем по представителю в классе эквивалентности



разбиение на классы эквивалентности

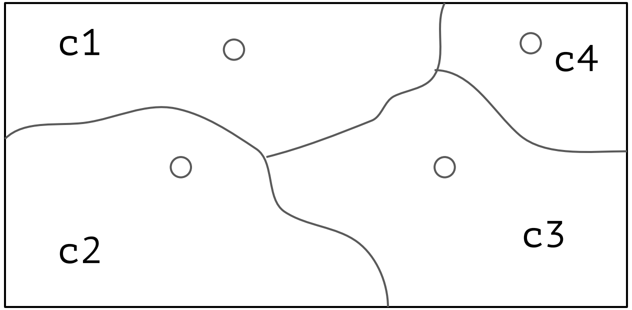
Union-find: реализация

Union-find (он же disjoint-set, он же структура данных для непересекающихся множеств)

Реализация (тривиальная):

1) Выбираем по представителю в классе эквивалентности

2) В каждый элемент добавляем ссылку на этого представителя



разбиение на классы эквивалентности

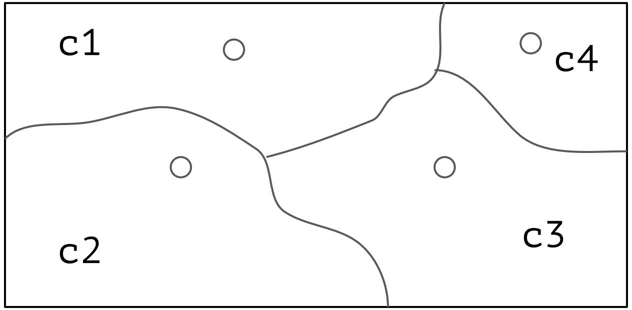
Union-find: реализация

Union-find (он же disjoint-set, он же структура данных для непересекающихся множеств)

Реализация (тривиальная):

1) Выбираем по представителю в классе эквивалентности

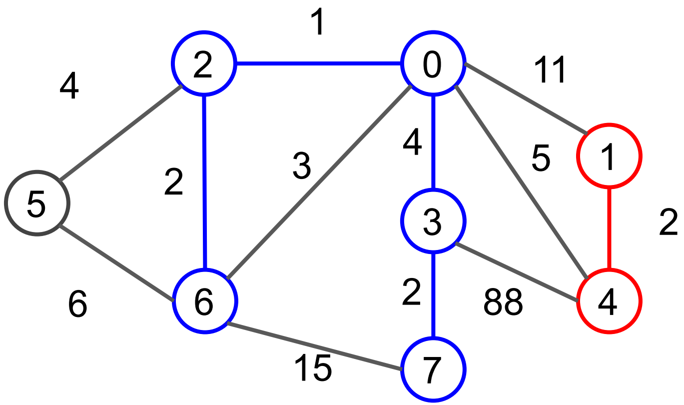
2) В каждый элемент добавляем ссылку на этого представителя



разбиение на классы эквивалентности

3) find(x) возвращает представителя
union(c1, c2) переписывает представителя одном из с

Алгоритм Краскала: union-find



0) Сортируем все ребра в порядке возрастания веса

1) Храним все вершины в union-find

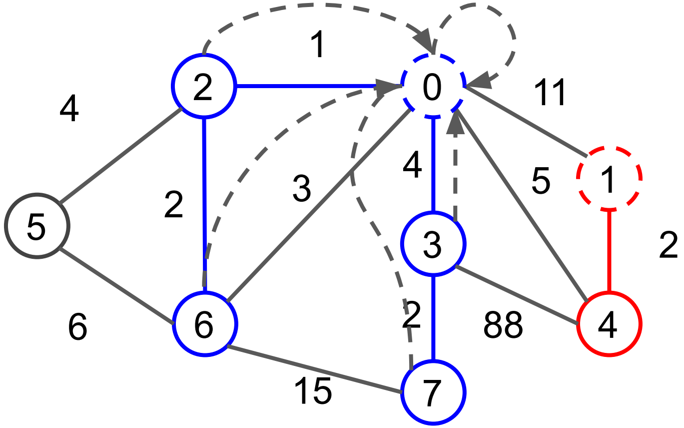
2) Идем по получившемуся массиву ребер и добавляем ребро в результат T, но только, вершины в разных классах эквивалентности!

берем ребро (u,v), только если find(u) != find(v)

3) При добавлении ребра сливаем классы экв. вершин

после этого делаем union(find(u), find(v))

Алгоритм Краскала: union-find



0) Сортируем все ребра в порядке возрастания веса

1) Храним все вершины в union-find

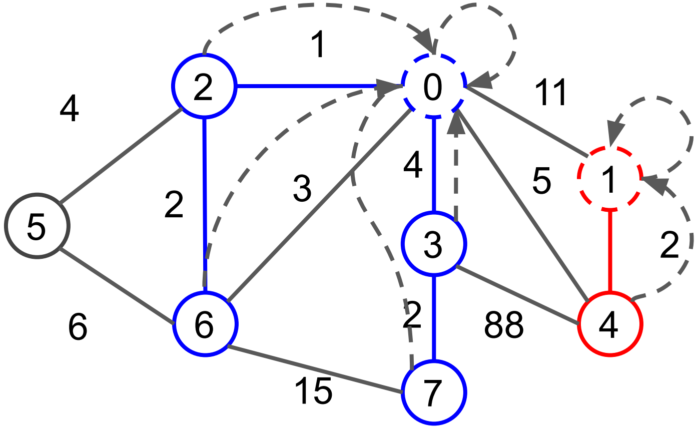
2) Идем по получившемуся массиву ребер и добавляем ребро в результат T, но только, вершины в разных классах эквивалентности!

берем ребро (u,v), только если find(u) != find(v)

3) При добавлении ребра сливаем классы экв. вершин

после этого делаем union(find(u), find(v))

Алгоритм Краскала: union-find



0) Сортируем все ребра в порядке возрастания веса

1) Храним все вершины в union-find

2) Идем по получившемуся массиву ребер и добавляем ребро в результат T, но только, вершины в разных классах эквивалентности!

берем ребро (u,v), только если find(u) != find(v)

3) При добавлении ребра сливаем классы экв. вершин

после этого делаем union(find(u), find(v))

Union-find: реализация

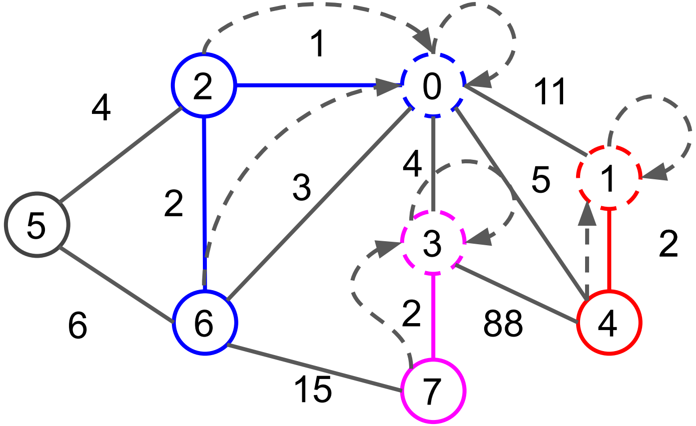
Реализация (на практике): union-find часто реализован просто, как массив (вне зависимости от политики реализации операций).

Пример:

Union-find: реализация

Реализация (на практике): union-find часто реализован просто, как массив (вне зависимости от политики реализации операций).

Пример:

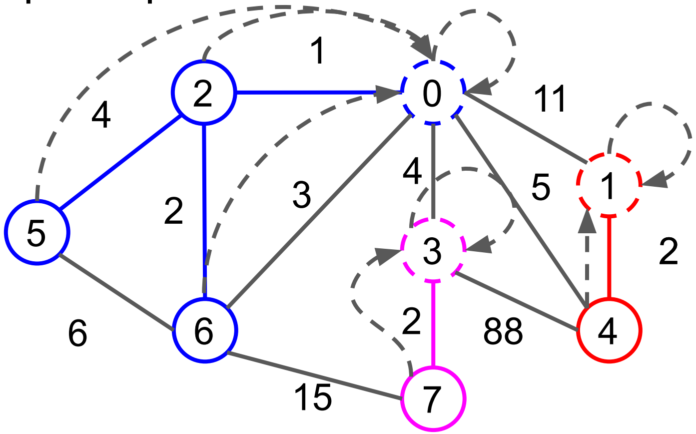


0	1	0	3	1	5	0	3
0	1	2	3	4	5	6	7

Union-find: реализация

Реализация (на практике): union-find часто реализован просто, как массив (вне зависимости от политики реализации операций).

Пример:

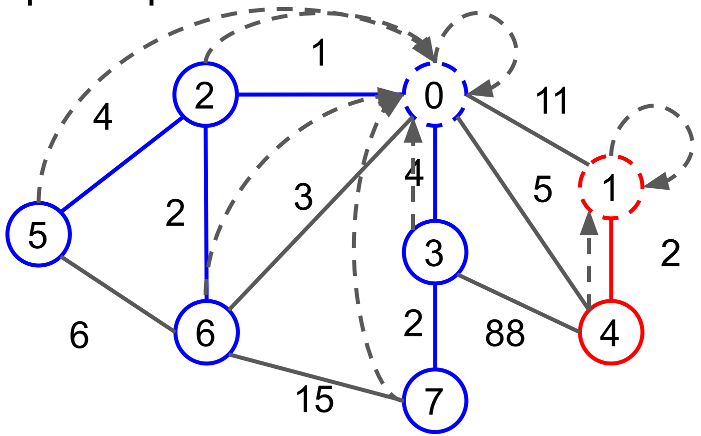


0	1	0	3	1	0	0	3
0	1	2	3	4	5	6	7

Union-find: реализация

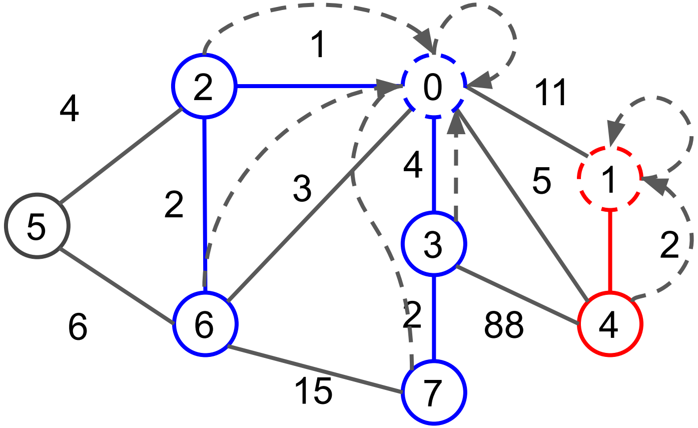
Реализация (на практике): union-find часто реализован просто, как массив (вне зависимости от политики реализации операций).

Пример:



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

Алгоритм Краскала: union-find



0) Сортируем все ребра в порядке возрастания веса

1) Храним все вершины в union-find

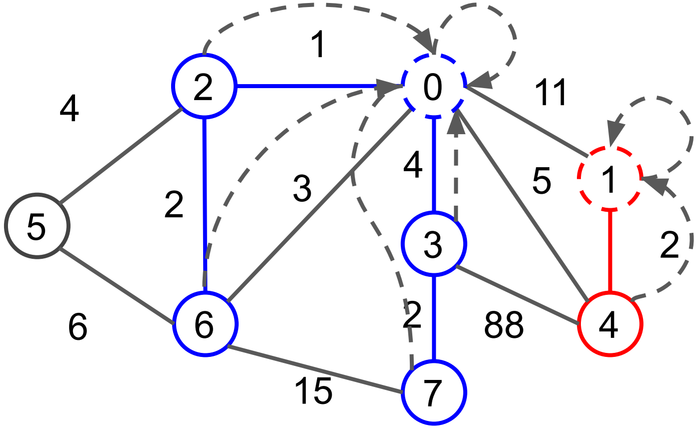
2) Идем по получившемуся массиву ребер и добавляем ребро в результат T, но только, вершины в разных классах эквивалентности!

берем ребро (u,v), только если find(u) != find(v)

3) При добавлении ребра сливаем классы экв. вершин

после этого делаем union(find(u), find(v))

Алгоритм Краскала: union-find



0) Сортируем все ребра в порядке возрастания веса

1) Храним все вершины в union-find

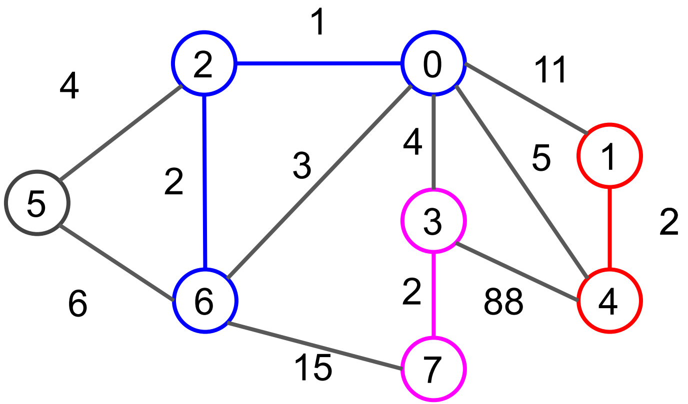
2) Идем по получившемуся массиву ребер и добавляем ребро в результат T, но только, вершины в разных классах эквивалентности!

берем ребро (u,v), только если
find(u) != find(v) ⟶ $O(1)$

3) При добавлении ребра сливаем классы экв. вершин

после этого делаем
union(find(u), find(v))

Алгоритм Краскала: union-find



Теперь это работает за O(1)!

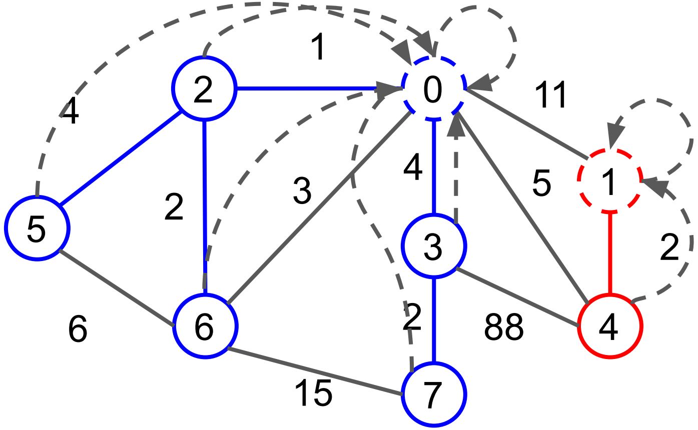
Сложность будет зависеть от реализации системы непересекающихся множеств

Наивная реализация: никакой системы, каждый раз просто проверяем на отсутствие циклов. Сложность?

На каждой из |E| итераций поиск цикла за |V|, что дает O(|V|*|E|).

Не забываем про сортировку ребер за O(|E|log(|E|)), уточняем до O(|V|log(|V|)) из-за $|E| \leq |V|^2$.

Алгоритм Краскала: union-find



0) Сортируем все ребра в порядке возрастания веса

1) Храним все вершины в union-find

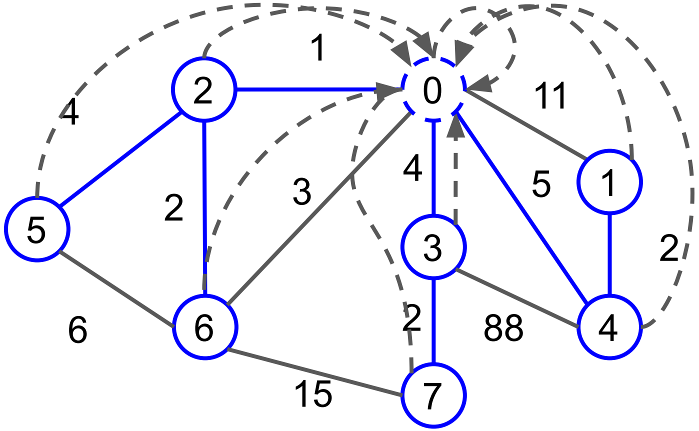
2) Идем по получившемуся массиву ребер и добавляем ребро в результат T, но только, вершины в разных классах эквивалентности!

берем ребро (u,v), только если
find(u) != find(v) $\longrightarrow O(1)$

3) При добавлении ребра сливаем классы экв. вершин

после этого делаем
union(find(u), find(v))

Алгоритм Краскала: union-find



0) Сортируем все ребра в порядке возрастания веса

1) Храним все вершины в union-find

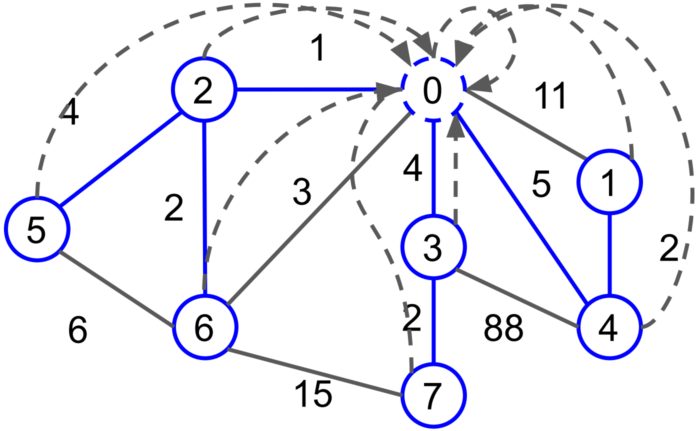
2) Идем по получившемуся массиву ребер и добавляем ребро в результат T, но только, вершины в разных классах эквивалентности!

берем ребро (u,v), только если find(u) != find(v) $\longrightarrow O(1)$

3) При добавлении ребра сливаем классы экв. вершин

после этого делаем union(find(u), find(v))

Алгоритм Краскала: union-find



0) Сортируем все ребра в порядке возрастания веса

1) Храним все вершины в union-find

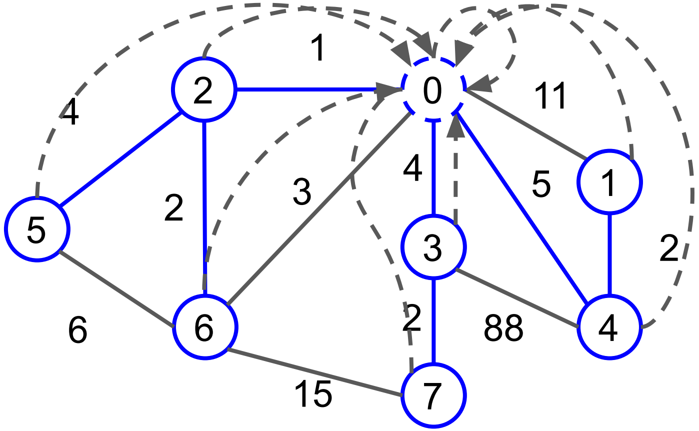
2) Идем по получившемуся массиву ребер и добавляем ребро в результат T, но только, вершины в разных классах эквивалентности!

берем ребро (u,v), только если find(u) != find(v) $\longrightarrow O(1)$

3) При добавлении ребра сливаем классы экв. вершин

$O(|V|) \longleftarrow$ после этого делаем union(find(u), find(v))

Алгоритм Краскала: union-find



0) Сортируем все ребра в порядке возрастания веса

1) Храним все вершины в union-find

2) Идем по получившемуся массиву ребер и добавляем ребро в результат T, но только, вершины в разных классах эквивалентности!

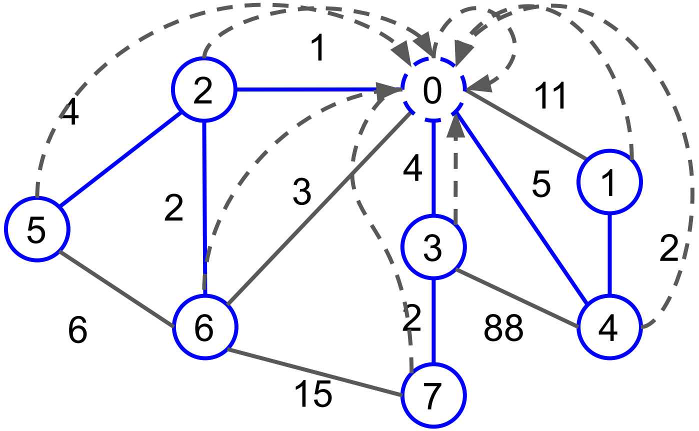
берем ребро (u,v), только если
find(u) != find(v) $\longrightarrow O(1)$

т.е. в сумме $O(|E|)$

3) При добавлении ребра сливаем классы экв. вершин

после этого делаем
union(find(u), find(v)) $\longrightarrow O(|V|)$

Алгоритм Краскала: union-find



0) Сортируем все ребра в порядке возрастания веса

1) Храним все вершины в union-find

2) Идем по получившемуся массиву ребер и добавляем ребро в результат T, но только, вершины в разных классах эквивалентности!

берем ребро (u,v), только если
find(u) != find(v) $\longrightarrow O(1)$

т.е. в сумме $O(|E|)$

3) При добавлении ребра сливаем классы экв. вершин

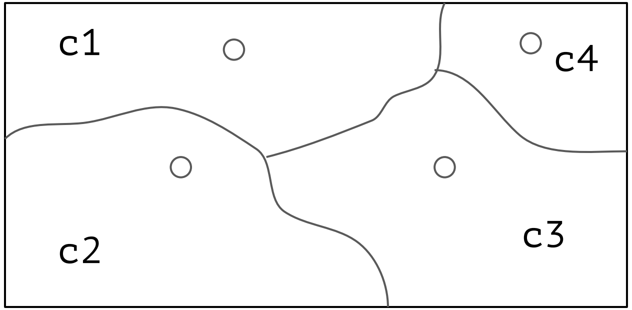
и еще: $O(|E| * |V|)$

$O(|V|) \longleftarrow$ union(find(u), find(v))

Алгоритм Краскала: union-find

Union-find (он же disjoint-set, он же структура данных для непересекающихся множеств)

Пока никак не помогает, все еще O(|V|*|E|).



разбиение на классы эквивалентности

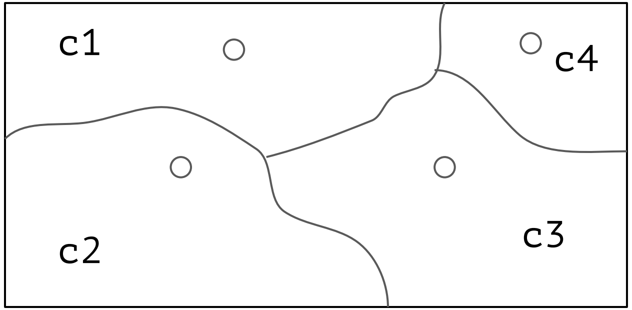
Алгоритм Краскала: union-find

Union-find (он же disjoint-set, он же структура данных для непересекающихся множеств)

Пока никак не помогает, все еще O(|V|*|E|).

Маленькая оптимизация :

Пусть при union мы всегда будем переписывать представителя меньшего класса, на представителя большего.



разбиение на классы эквивалентности

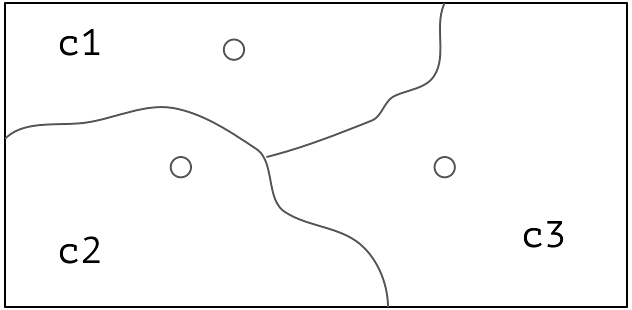
Алгоритм Краскала: union-find

Union-find (он же disjoint-set, он же структура данных для непересекающихся множеств)

Пока никак не помогает, все еще O(|V|*|E|).

Маленькая оптимизация :

Пусть при union мы всегда будем переписывать представителя меньшего класса, на представителя большего.



разбиение на классы эквивалентности

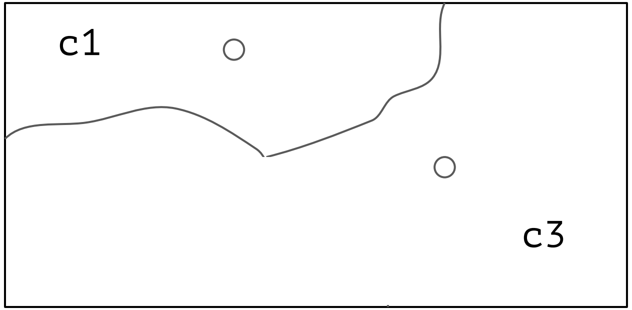
Алгоритм Краскала: union-find

Union-find (он же disjoint-set, он же структура данных для непересекающихся множеств)

Пока никак не помогает, все еще O(|V|*|E|).

Маленькая оптимизация :

Пусть при union мы всегда будем переписывать представителя меньшего класса, на представителя большего.



разбиение на классы эквивалентности

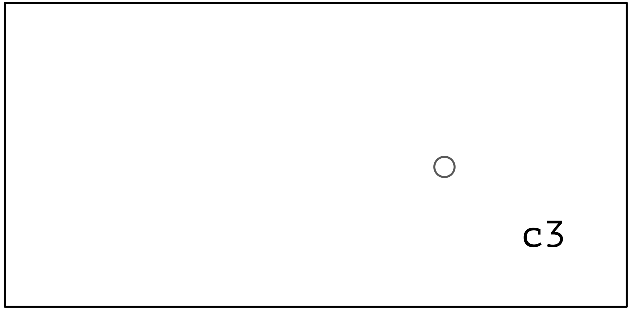
Алгоритм Краскала: union-find

Union-find (он же disjoint-set, он же структура данных для непересекающихся множеств)

Пока никак не помогает, все еще O(|V|*|E|).

Маленькая оптимизация :

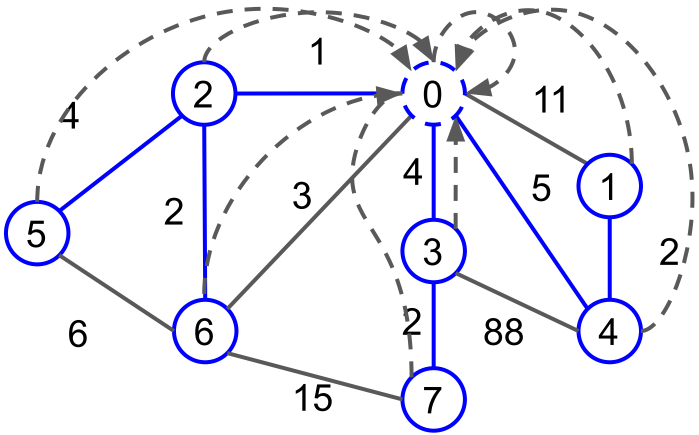
Пусть при union мы всегда будем переписывать представителя меньшего класса, на представителя большего.



разбиение на классы эквивалентности

Этого легко добиться, сохраняя размер в представителе.

Алгоритм Краскала: union-find



0) Сортируем все ребра в порядке возрастания веса

1) Храним все вершины в union-find

2) Идем по получившемуся массиву ребер и добавляем ребро в результат T, но только, вершины в разных классах эквивалентности!

берем ребро (u,v), только если
find(u) != find(v) $\longrightarrow O(1)$

т.е. в сумме $O(|E|)$

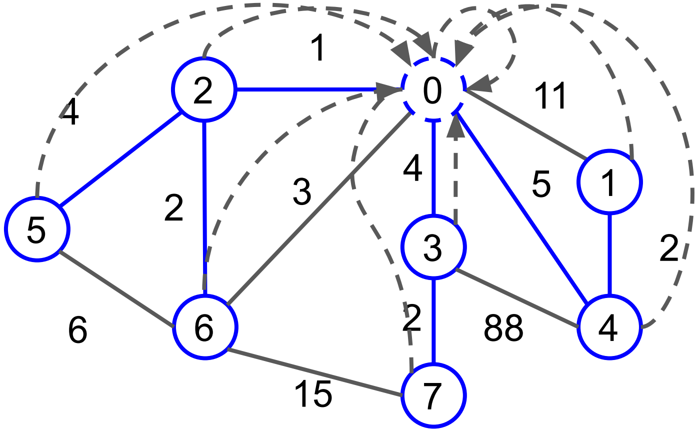
3) При добавлении ребра сливаем классы экв. вершин

и еще: $O(|E| * |V|)$

$O(|V|) \longleftarrow$ union(find(u), find(v))

Алгоритм Краскала: union-find

Все еще все очень плохо!
Слияние все еще за линию!



0) Сортируем все ребра в порядке возрастания веса

1) Храним все вершины в union-find

2) Идем по получившемуся массиву ребер и добавляем ребро в результат T, но только, вершины в разных классах эквивалентности!

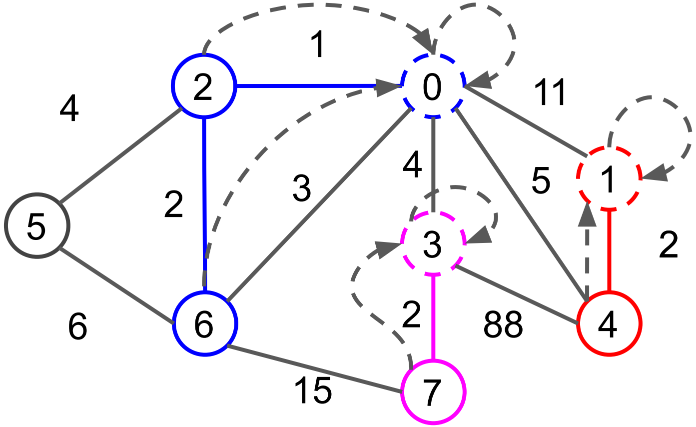
3) При добавлении ребра сливаем классы экв. вершин

берем ребро (u,v), только если
find(u) != find(v) $\longrightarrow O(1)$

т.е. в сумме $O(|E|)$

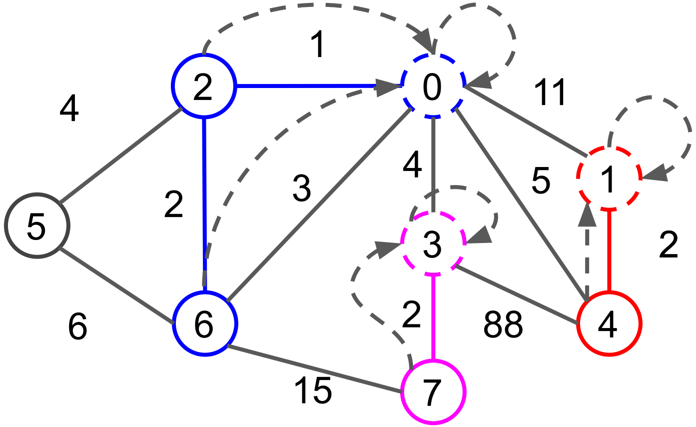
$\boxed{\text{и еще: } O(|E| * |V|) \quad O(|V|) \longleftarrow \texttt{union(find(u), find(v))}}$

Алгоритм Краскала: union-find



Но! Давайте посмотрим на все с точки зрения каждой из вершин графа.

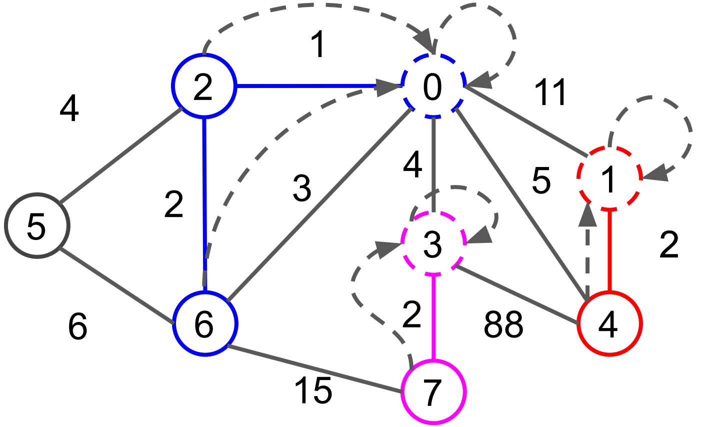
Алгоритм Краскала: union-find



Но! Давайте посмотрим на все с точки зрения каждой из вершин графа.

Сколько раз в худшем случае для каждой вершины перепишут представителя с учетом оптимизации?

Алгоритм Краскала: union-find

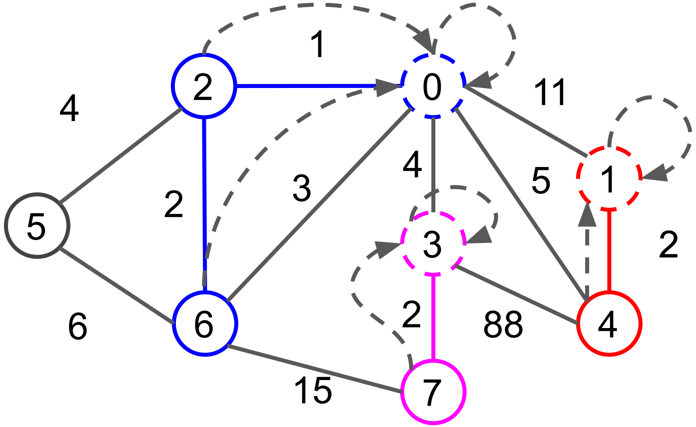


Но! Давайте посмотрим на все с точки зрения каждой из вершин графа.

Сколько раз в худшем случае для каждой вершины перепишут представителя с учетом оптимизации?

Если нам переписывают приоритет, то мы точно попадаем в новый класс эквивалентности, который как минимум равен нашему.

Алгоритм Краскала: union-find

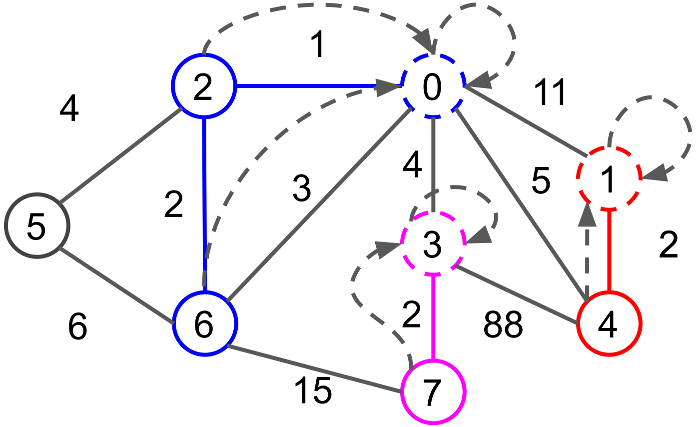


Но! Давайте посмотрим на все с точки зрения каждой из вершин графа.

Сколько раз в худшем случае для каждой вершины перепишут представителя с учетом оптимизации?

Если нам переписывают приоритет, то мы точно попадаем в новый класс эквивалентности, который как минимум равен нашему => в самом худшем случае класс эквивалентности всегда удваивается

Алгоритм Краскала: union-find

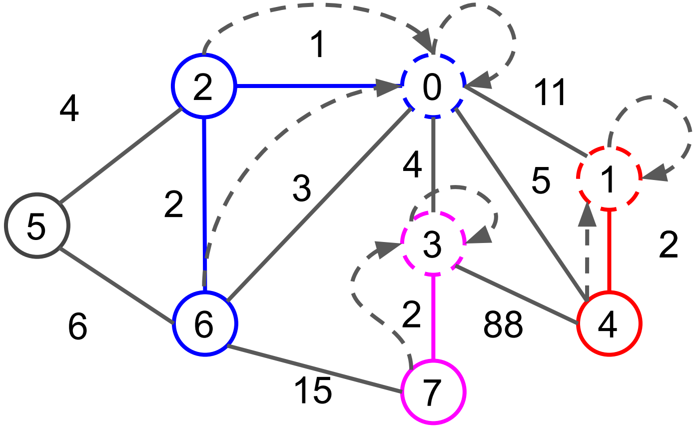


Но! Давайте посмотрим на все с точки зрения каждой из вершин графа.

Сколько раз в худшем случае для каждой вершины перепишут представителя с учетом оптимизации?

Если нам переписывают приоритет, то мы точно попадаем в новый класс эквивалентности, который как минимум равен нашему => в самом худшем случае класс эквивалентности всегда удваивается => делать так можно не больше, чем log(|V|).

Алгоритм Краскала: union-find



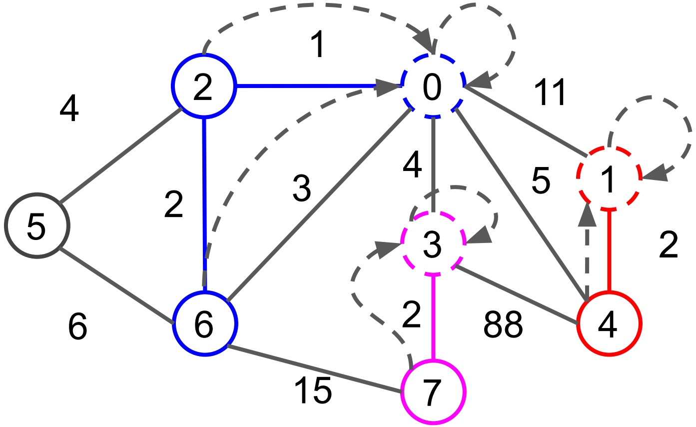
Итого:

1) O(|E|*log(|V|)) за сортировку

2) O(|E|) за проверку ребер на циклы

3) O(|V|*log(|V|)) за union-ы

Алгоритм Краскала: union-find



Итого:

1) O(|E|*log(|V|)) за сортировку

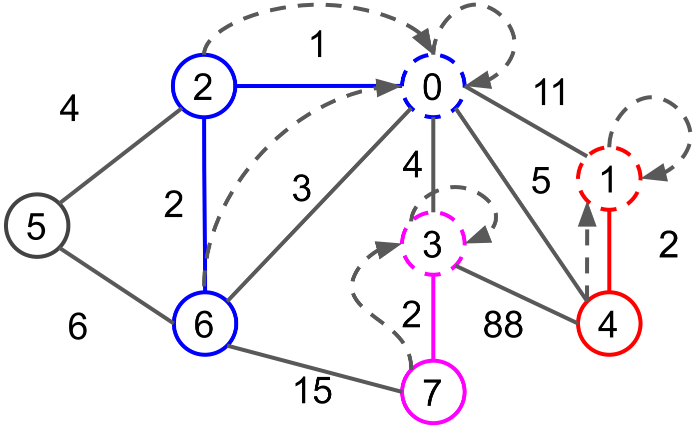
2) O(|E|) за проверку ребер на циклы

3) O(|V|*log(|V|)) за union-ы

Что здесь на самом деле произошло: хотя каждая операция union может занимать линейное время в худшем случае, т.е. дает честное O(|V|), сложность последовательности union-ов дает всего O(|V|*log(|V|))!

Как такое называется?

Алгоритм Краскала: union-find



Итого:

1) O(|E|*log(|V|)) за сортировку

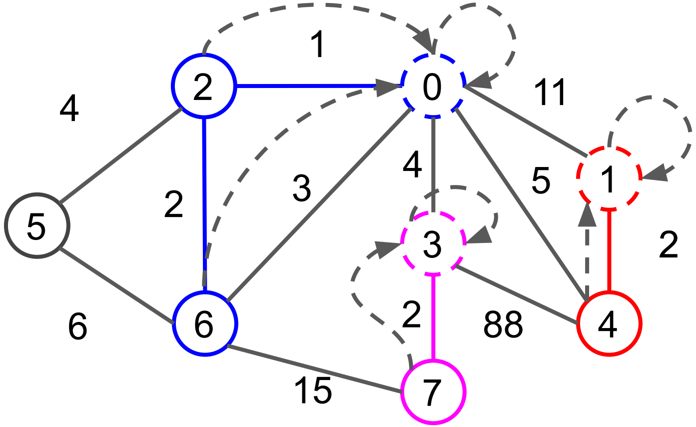
2) O(|E|) за проверку ребер на циклы

3) O(|V|*log(|V|)) за union-ы

Что здесь на самом деле произошло: хотя каждая операция union может занимать линейное время в худшем случае, т.е. дает честное O(|V|), сложность последовательности union-ов дает всего O(|V|*log(|V|))!

Как такое называется? Амортизационная сложность! (в этот раз без монет)

Алгоритм Краскала: union-find



Итого:

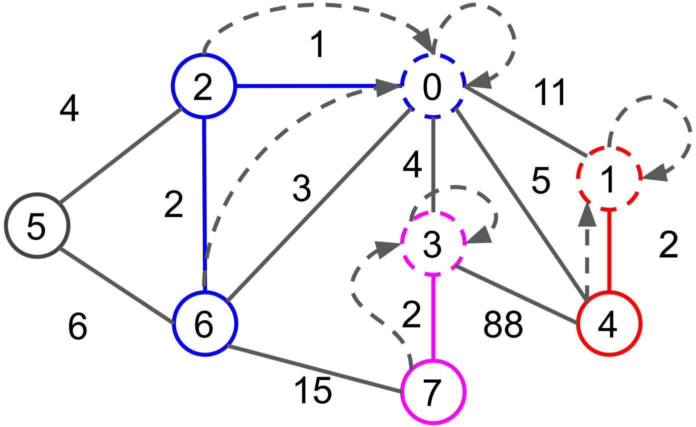
1) O(|E|*log(|V|)) за сортировку

2) O(|E|) за проверку ребер на циклы

3) ~~O(|V|*log(|V|))~~ за union-ы
O(|E|*log(|V|)) из-за связности графа

Догнали Приму!

Алгоритм Краскала: union-find



Итого:

1) O(|E|*log(|V|)) за сортировку

2) O(|E|) за проверку ребер на циклы

3) ~~O(|V|*log(|V|))~~ за union-ы
O(|E|*log(|V|)) из-за связности графа

Но можем ли мы еще лучше?

Догнали Приму!

Union-find: реализация #2

Реализация (ленивая):

1) Пусть теперь union не переписывает всем элементам одного из множеств его представителя

Union-find: реализация #2

Реализация (ленивая):

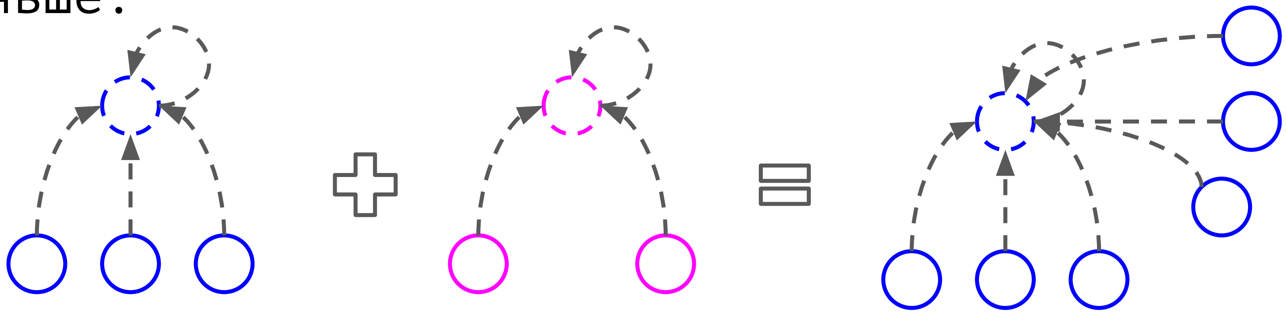
1) Пусть теперь union не переписывает всем элементам одного из множеств его представителя. Теперь он сделает это только для старого представителя.

Union-find: реализация #2

Реализация (ленивая):

1) Пусть теперь union не переписывает всем элементам одного из множеств его представителя. Теперь он сделает это только для старого представителя.

Раньше:

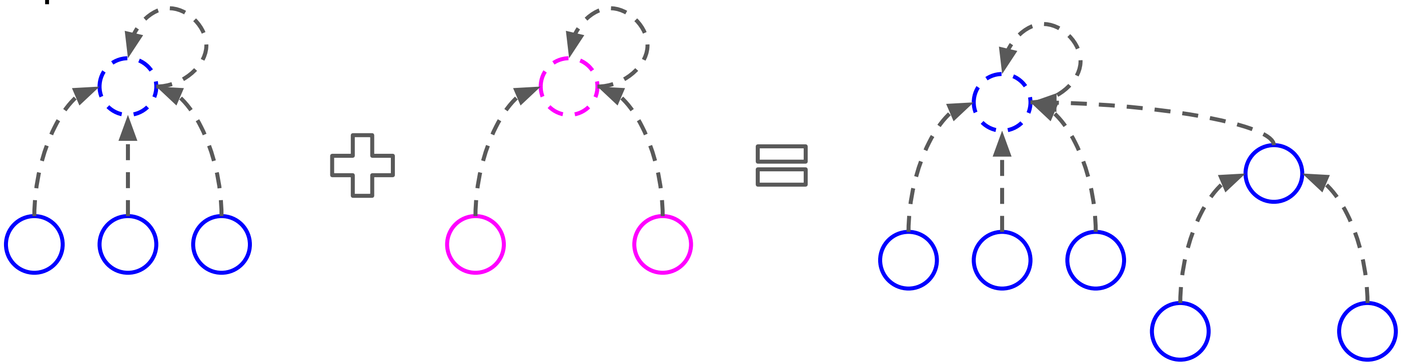


Union-find: реализация #2

Реализация (ленивая):

1) Пусть теперь union не переписывает всем элементам одного из множеств его представителя. Теперь он сделает это только для старого представителя.

Теперь:



Union-find: реализация #2

Реализация (ленивая):

1) Пусть теперь union не переписывает всем элементам одного из множеств его представителя. Теперь он сделает это только для старого представителя.

Если на руках представители (корни), то такая операция работает за O(1).

Union-find: реализация #2

Реализация (ленивая):

1) Пусть теперь union не переписывает всем элементам одного из множеств его представителя. Теперь он сделает это только для старого представителя.

Если на руках представители (корни), то такая операция работает за O(1). Но обычно то у нас какие-то два элемента, поэтому union сначала вызывает find от каждого из элементов, а уж потом объединяет их за O(1).

Union-find: реализация #2

Реализация (ленивая):

1) Пусть теперь union не переписывает всем элементам одного из множеств его представителя. Теперь он сделает это только для старого представителя.

2) Но как теперь работает find?

Union-find: реализация #2

Реализация (ленивая):

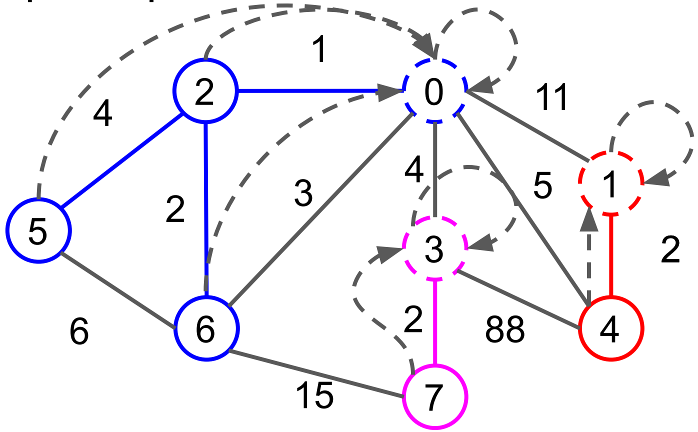
1) Пусть теперь union не переписывает всем элементам одного из множеств его представителя. Теперь он сделает это только для старого представителя.

2) Но как теперь работает find? Сразу ответ дать не можем, нужна итерация: проходим по ссылкам до того момента, пока представитель элемента не станет равен ему самому.

Union-find: реализация

Реализация (на практике): union-find часто реализован просто, как массив (вне зависимости от политики реализации операций).

Пример:

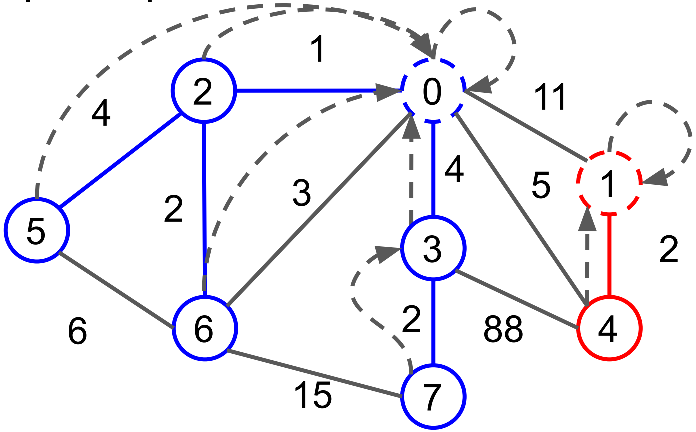


0	1	0	3	1	0	0	3
0	1	2	3	4	5	6	7

Union-find: реализация

Реализация (на практике): union-find часто реализован просто, как массив (вне зависимости от политики реализации операций).

Пример:

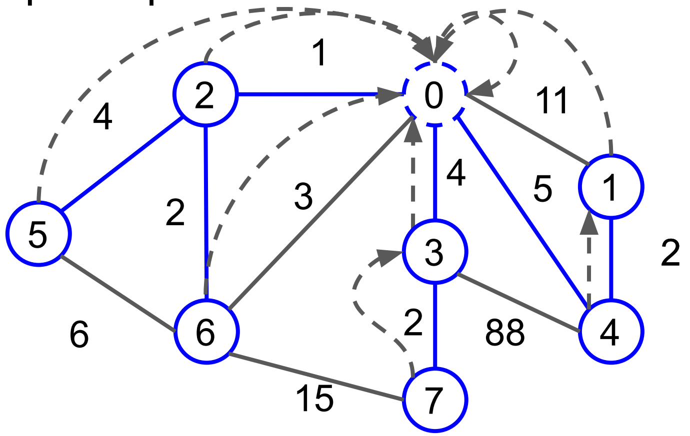


0	1	0	0	1	0	0	**3**
0	1	2	3	4	5	6	7

Union-find: реализация

Реализация (на практике): union-find часто реализован просто, как массив (вне зависимости от политики реализации операций).

Пример:

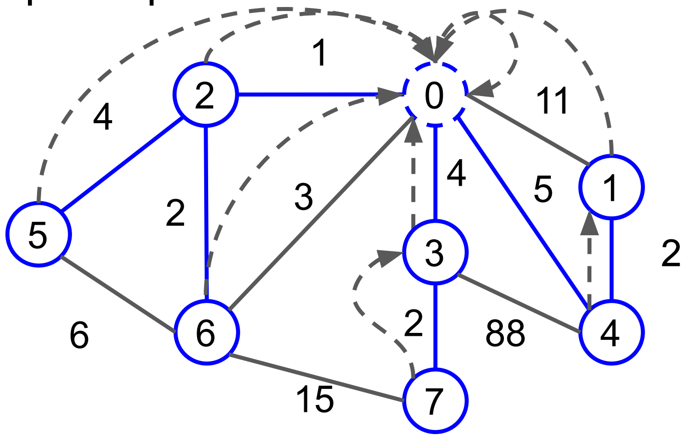


0	0	0	0	**1**	0	0	**3**
0	1	2	3	4	5	6	7

Union-find: реализация

Реализация (на практике): union-find часто реализован просто, как массив (вне зависимости от политики реализации операций).

Пример:



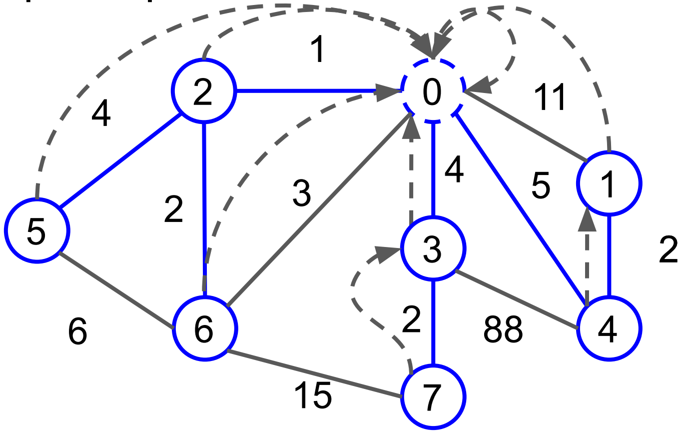
0	1	2	3	4	5	6	7
0	0	0	0	**1**	0	0	**3**

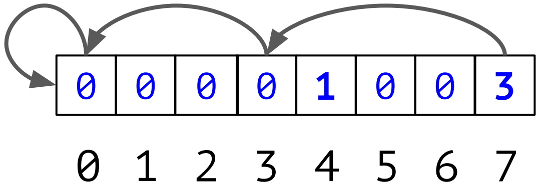
find(7)?

Union-find: реализация

Реализация (на практике): union-find часто реализован просто, как массив (вне зависимости от политики реализации операций).

Пример:





find(7)?

Union-find: реализация #2

Реализация (ленивая):

1) Пусть теперь union не переписывает всем элементам одного из множеств его представителя. Теперь он сделает это только для старого представителя.

2) Но как теперь работает find? Сразу ответ дать не можем, нужна итерация: проходим по ссылкам до того момента, пока представитель элемента не станет равен ему самому.

Union-find: реализация #2

Реализация (ленивая):

1) Пусть теперь union не переписывает всем элементам одного из множеств его представителя. Теперь он сделает это только для старого представителя. $\longrightarrow O(n)$

2) Но как теперь работает find? Сразу ответ дать не можем, нужна итерация: проходим по ссылкам до того момента, пока представитель элемента не станет равен ему самому. $\longrightarrow O(n)$

Union-find: реализация #2

Реализация (ленивая):

1) Пусть теперь union не переписывает всем элементам одного из множеств его представителя. Теперь он сделает это только для старого представителя. $\longrightarrow O(n)$

2) Но как теперь работает find? Сразу ответ дать не можем, нужна итерация: проходим по ссылкам до того момента, пока представитель элемента не станет равен ему самому. $\longrightarrow O(n)$

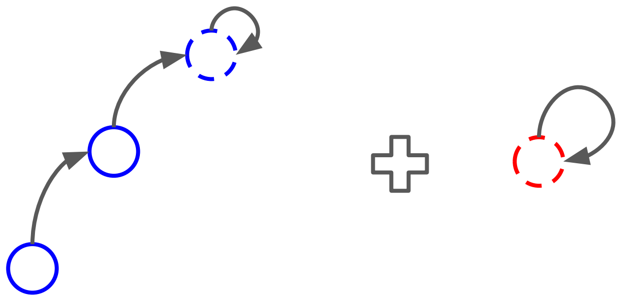
Чтобы работало хорошо, нужны оптимизации.

Union-find: объединение по рангу

Проблема опять в несбалансированности (аналогия с BST)

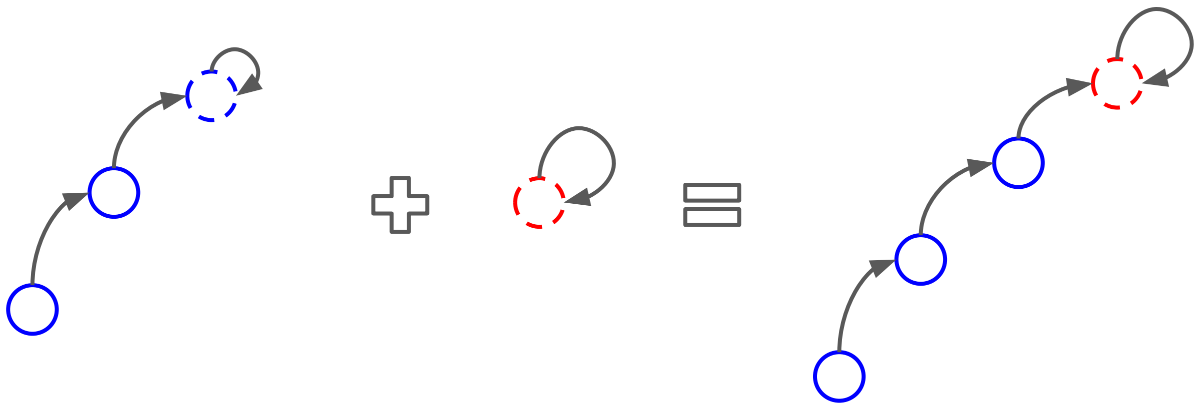
Union-find: объединение по рангу

Проблема опять в несбалансированности (аналогия с BST)



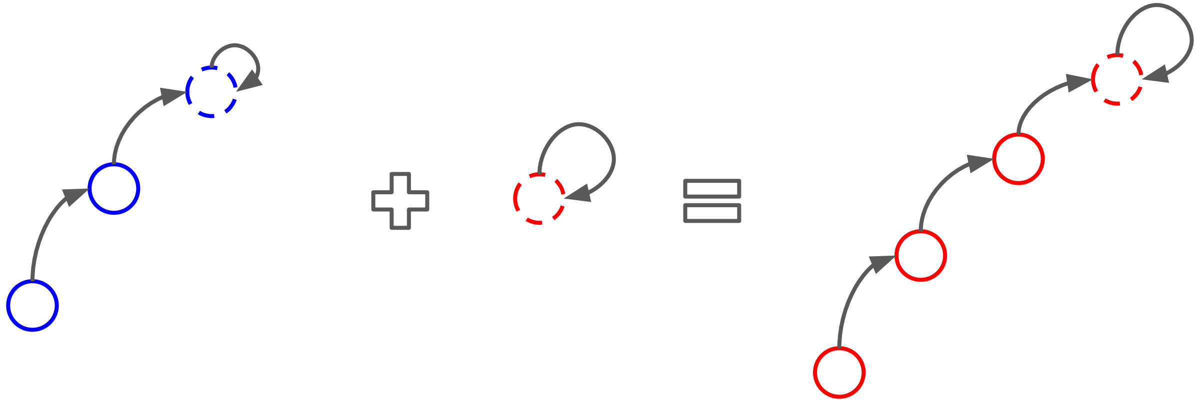
Union-find: объединение по рангу

Проблема опять в несбалансированности (аналогия с BST)



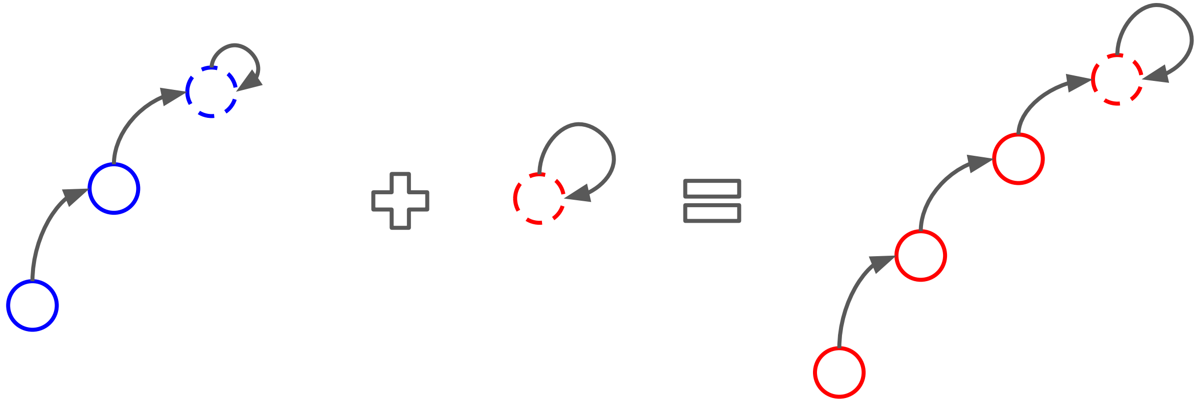
Union-find: объединение по рангу

Проблема опять в несбалансированности (аналогия с BST)



Union-find: объединение по рангу

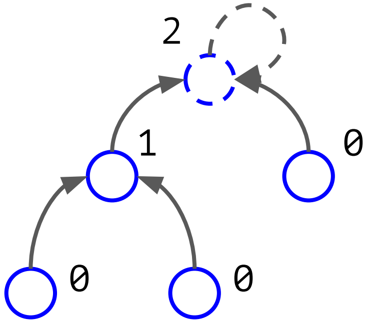
Проблема опять в несбалансированности (аналогия с BST)
Чтобы ее исправить вводятся ранги для каждой вершины.



Union-find: объединение по рангу

Проблема опять в несбалансированности (аналогия с BST)
Чтобы ее исправить вводятся ранги для каждой вершины.

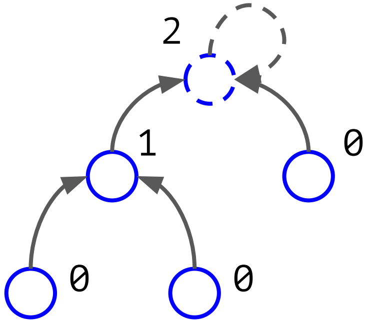
$rank[x]$ - высота поддерева, растущего из x.



Union-find: объединение по рангу

Проблема опять в несбалансированности (аналогия с BST)
Чтобы ее исправить вводятся ранги для каждой вершины.

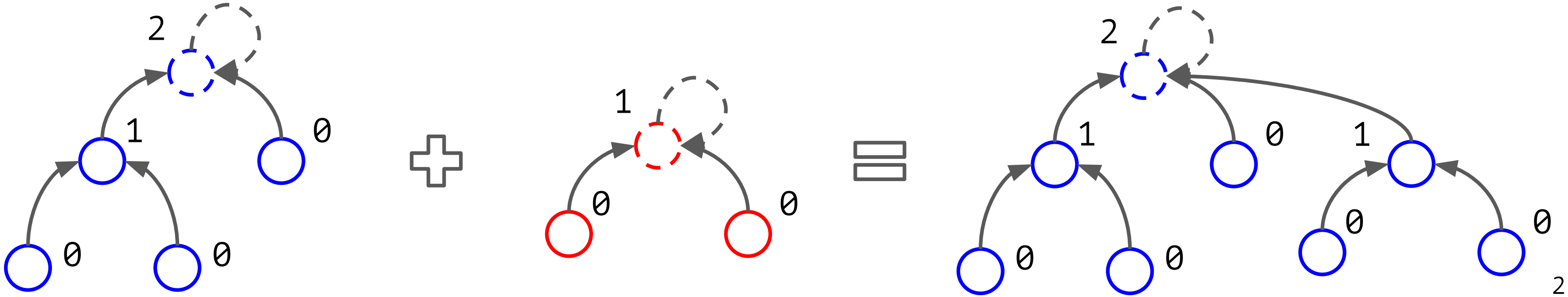
$rank[x]$ - высота поддерева, растущего из x.
Правило: при union, подвязываем дерево с меньшим рангом корня к дереву с большим



Union-find: объединение по рангу

Проблема опять в несбалансированности (аналогия с BST)
Чтобы ее исправить вводятся ранги для каждой вершины.

$rank[x]$ - высота поддерева, растущего из x.
Правило: при union, подвязываем дерево с меньшим рангом корня к дереву с большим



Union-find: объединение по рангу

Формальнее:

```
def union(x, y):
  a = find(x), b = find(y)
  if rank[a] > rank[b]:
    parent[b] = a
  else:
    parent[a] = b
```

Union-find: объединение по рангу

Формальнее:

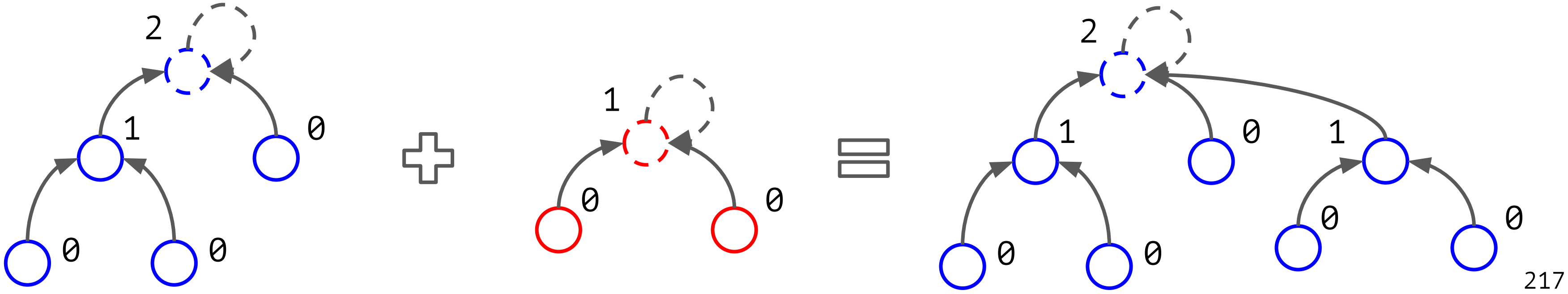
```
def union(x, y):
  a = find(x), b = find(y)
  if rank[a] > rank[b]:
    parent[b] = a
  else:
    parent[a] = b
```

Ничего ли мы
не забыли?

Union-find: объединение по рангу

Проблема опять в несбалансированности (аналогия с BST)
Чтобы ее исправить вводятся ранги для каждой вершины.

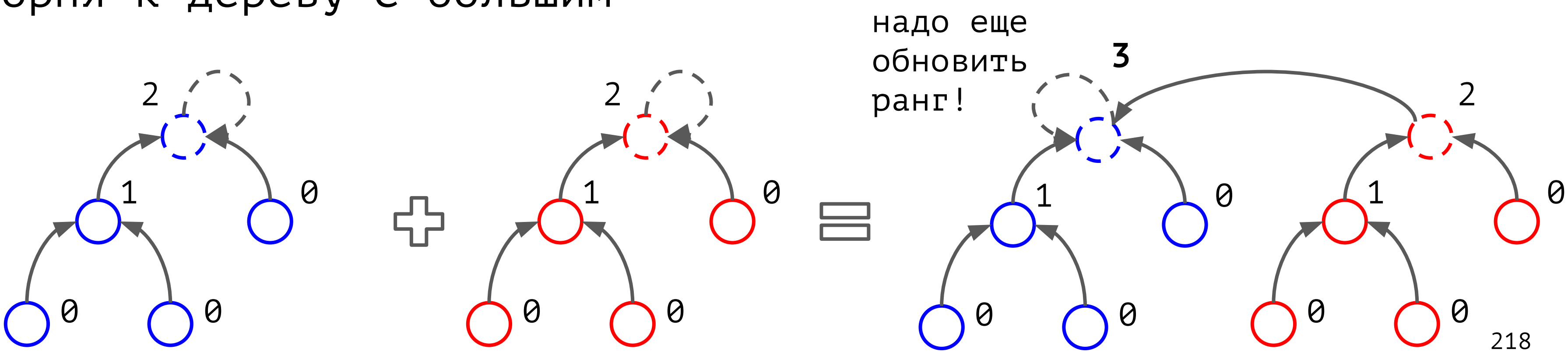
$rank[x]$ - высота поддерева, растущего из x.
Правило: при union, подвязываем дерево с меньшим рангом корня к дереву с большим



Union-find: объединение по рангу

Проблема опять в несбалансированности (аналогия с BST)
Чтобы ее исправить вводятся ранги для каждой вершины.

$rank[x]$ - высота поддерева, растущего из x.
Правило: при union, подвязываем дерево с меньшим рангом корня к дереву с большим



Union-find: объединение по рангу

Формальнее:

```
def union(x, y):
  a = find(x), b = find(y)
  if rank[a] > rank[b]:
    parent[b] = a
  else:
    parent[a] = b
    if rank[a] == rank[b]:
        rank[b] += 1
```

Ничего ли мы
не забыли?

Обновляем ранг

Union-find: объединение по рангу (анализ)

Утверждение: в ленивой реализации union-find с объединением по рангу сложность операций union и find в худшем случае составляет $O(logN)$, где N - количество элементов в структуре данных.

Union-find: объединение по рангу (анализ)

Утверждение: в ленивой реализации union-find с объединением по рангу сложность операций union и find в худшем случае составляет $O(logN)$, где N - количество элементов в структуре данных.

Лемма о рангах: пусть в union-find есть N элементов, тогда для любого $r \geq 0$ верно, что объектов с рангом r не больше, чем $\frac{N}{2^r}$.

Union-find: объединение по рангу (анализ)

Утверждение: в ленивой реализации union-find с объединением по рангу сложность операций union и find в худшем случае составляет $O(logN)$, где N - количество элементов в структуре данных.

Лемма о рангах: пусть в union-find есть N элементов, тогда для любого $r \geq 0$ верно, что объектов с рангом r не больше, чем $\frac{N}{2^r}$.

Следствие: максимальный ранг $\leq log_2 N$

Union-find: объединение по рангу (анализ)

Утверждение: в ленивой реализации union-find с объединением по рангу сложность операций union и find в худшем случае составляет $O(logN)$, где N - количество элементов в структуре данных.

Лемма о рангах: пусть в union-find есть N элементов, тогда для любого $r \geq 0$ верно, что объектов с рангом r не больше, чем $\frac{N}{2^r}$.

Следствие: максимальный ранг $\leq log_2 N$

(в качестве r берем $log_2 N$ => с таким рангом может быть не больше одного, а дальше только хуже)

Union-find: объединение по рангу (анализ)

Утверждение: в ленивой реализации union-find с объединением по рангу сложность операций union и find в худшем случае составляет $O(logN)$, где N - количество элементов в структуре данных.

Лемма о рангах: пусть в union-find есть N элементов, тогда для любого $r \geq 0$ верно, что объектов с рангом r не больше, чем $\frac{N}{2^r}$.

Следствие: максимальный ранг $\leq log_2 N$

(в качестве r берем $log_2 N$ => с таким рангом может быть не больше одного, а дальше только хуже)

Union-find: объединение по рангу (анализ)

Лемма о рангах: пусть в union-find есть N элементов, тогда для любого $r \geq 0$ верно, что объектов с рангом r не больше, чем $\frac{N}{2^r}$.

Доказательство (схематично):

Union-find: объединение по рангу (анализ)

Лемма о рангах: пусть в union-find есть N элементов, тогда для любого $r \geq 0$ верно, что объектов с рангом r не больше, чем $\frac{N}{2^r}$.

Доказательство (схематично):

1) по индукции доказываем, что в поддереве, у корня которого ранг r, будет $\geq 2^r$ элементов

2) замечаем, что поддеревья, у корней которых совпадают ранги, не пересекаются

Union-find: объединение по рангу (анализ)

Лемма о рангах: пусть в union-find есть N элементов, тогда для любого $r \geq 0$ верно, что объектов с рангом r не больше, чем $\frac{N}{2^r}$.

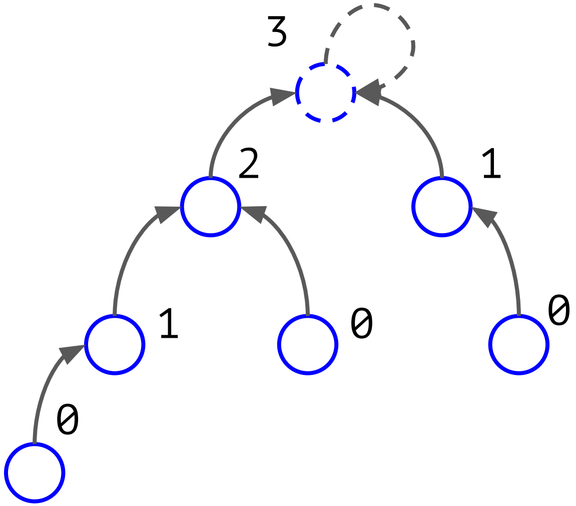
Доказательство (схематично):

1) по индукции доказываем, что в поддереве, у корня которого ранг r, будет $\geq 2^r$ элементов

2) замечаем, что поддеревья, у корней которых совпадают ранги, не пересекаются

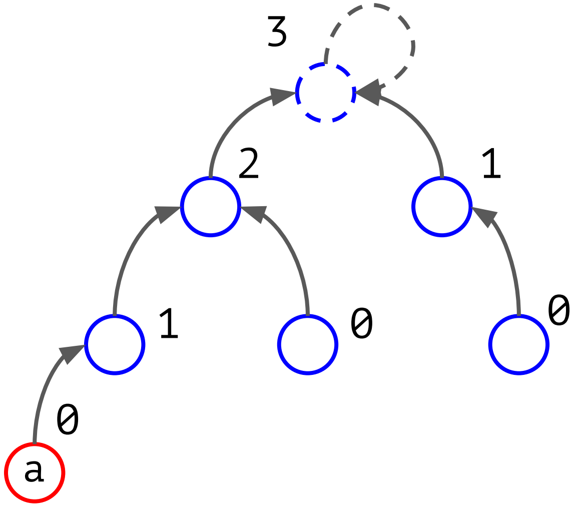
3) получаем оценку на количество таких деревьев

Union-find: сжатие путей



Как еще разогнать поиск в `union-find`?

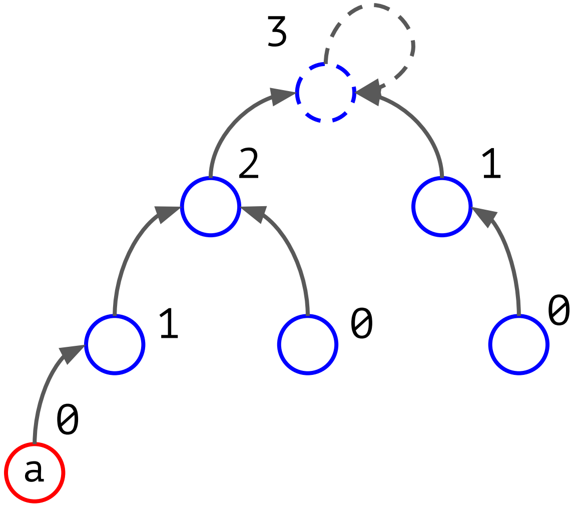
Union-find: сжатие путей



Как еще разогнать поиск в union-find?

Неприятная ситуация: когда раз за разом ищем представителя для нижнего элемента

Union-find: сжатие путей



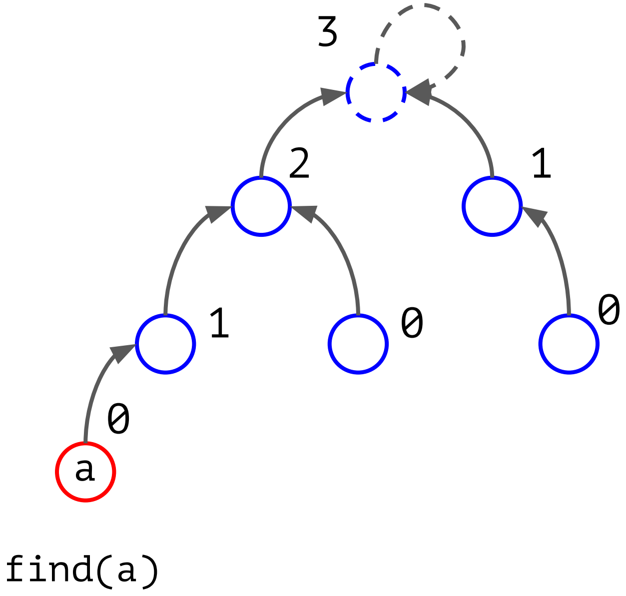
Как еще разогнать поиск в union-find?

Неприятная ситуация: когда раз за разом ищем представителя для нижнего элемента

Попробуем оптимизировать!

Пусть теперь find после своей работы обновляет представителя того, от кого его позвали.

Union-find: сжатие путей



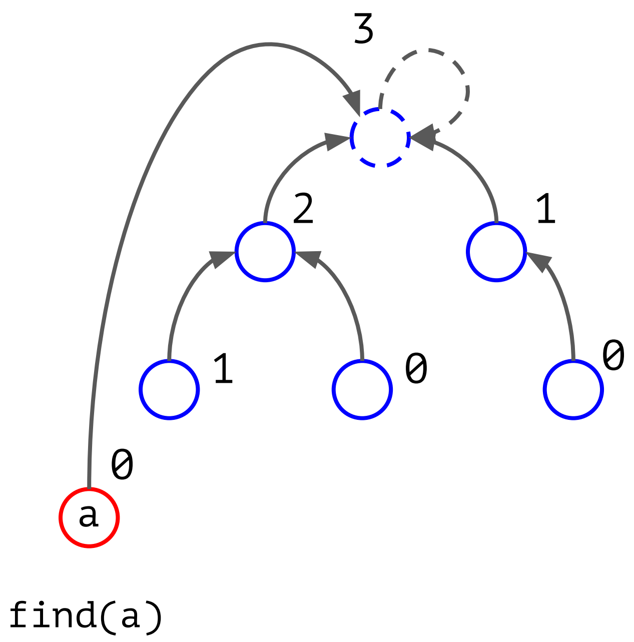
Как еще разогнать поиск в union-find?

Неприятная ситуация: когда раз за разом ищем представителя для нижнего элемента

Попробуем оптимизировать!

Пусть теперь find после своей работы обновляет представителя того, от кого его позвали.

Union-find: сжатие путей



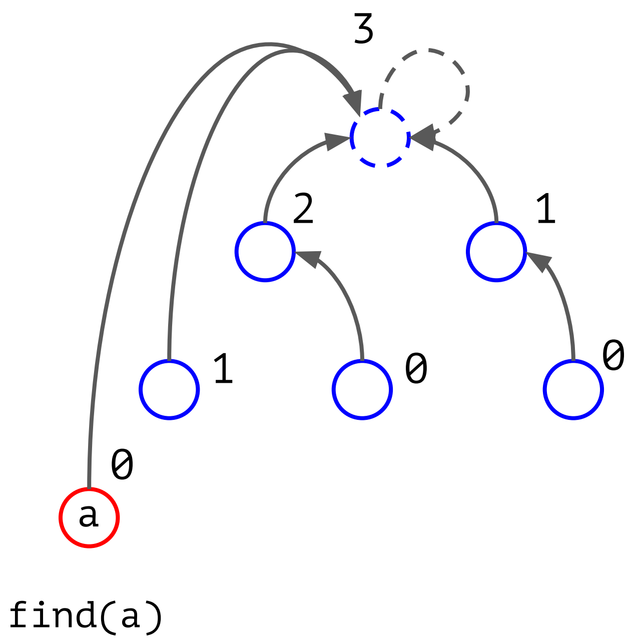
Как еще разогнать поиск в union-find?

Неприятная ситуация: когда раз за разом ищем представителя для нижнего элемента

Попробуем оптимизировать!

Пусть теперь find после своей работы обновляет представителя того, от кого его позвали.

Union-find: сжатие путей



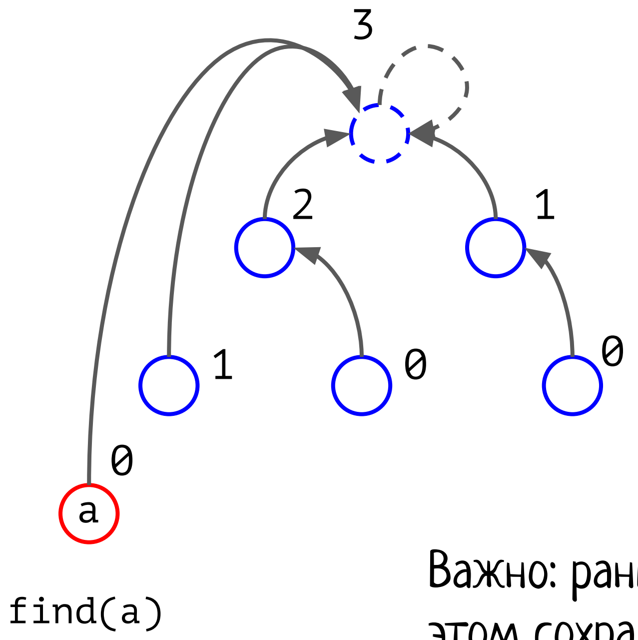
Как еще разогнать поиск в union-find?

Неприятная ситуация: когда раз за разом ищем представителя для нижнего элемента

Попробуем оптимизировать!

Пусть теперь find после своей работы обновляет представителя того, от кого его позвали.
(и всех по пути)

Union-find: сжатие путей



Важно: ранги при этом сохраняются!

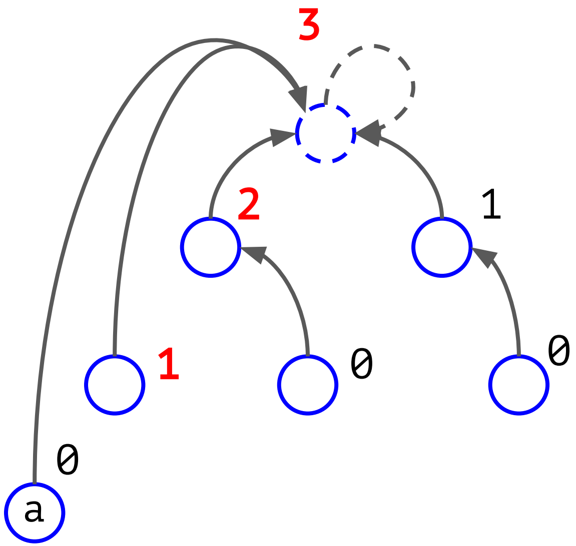
Как еще разогнать поиск в union-find?

Неприятная ситуация: когда раз за разом ищем представителя для нижнего элемента

Попробуем оптимизировать!

Пусть теперь find после своей работы обновляет представителя того, от кого его позвали. (и всех по пути)

Union-find: сжатие путей



Важно: ранги при этом сохраняются!

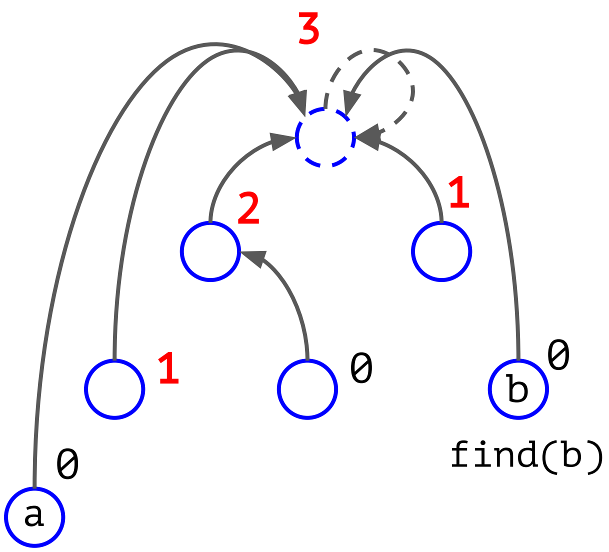
Как еще разогнать поиск в union-find?

Неприятная ситуация: когда раз за разом ищем представителя для нижнего элемента

Попробуем оптимизировать!

Пусть теперь find после своей работы обновляет представителя того, от кого его позвали.

Union-find: сжатие путей



Важно: ранги при этом сохраняются!

Как еще разогнать поиск в union-find?

Неприятная ситуация: когда раз за разом ищем представителя для нижнего элемента

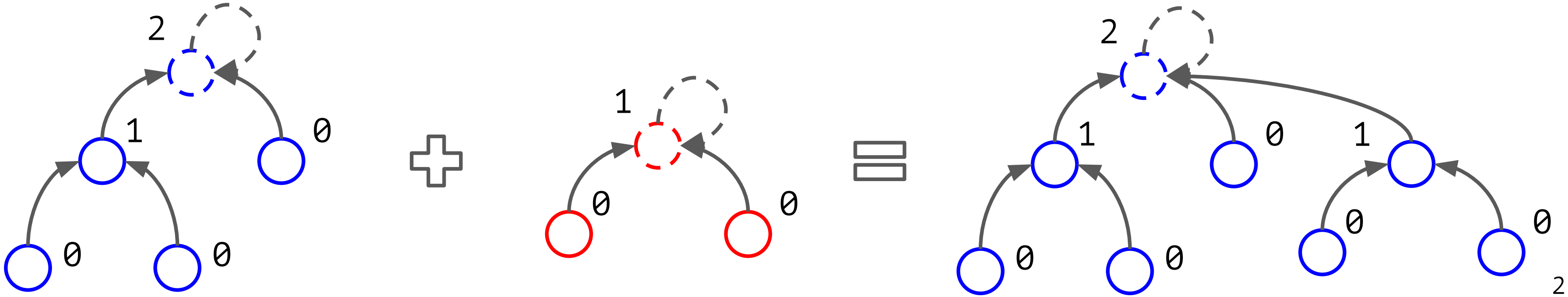
Попробуем оптимизировать!

Пусть теперь find после своей работы обновляет представителя того, от кого его позвали.

Union-find: объединение по рангу

$rank[x]$ - высота поддерева, растущего из x.

Правило: при union, подвязываем дерево с меньшим рангом корня к дереву с большим

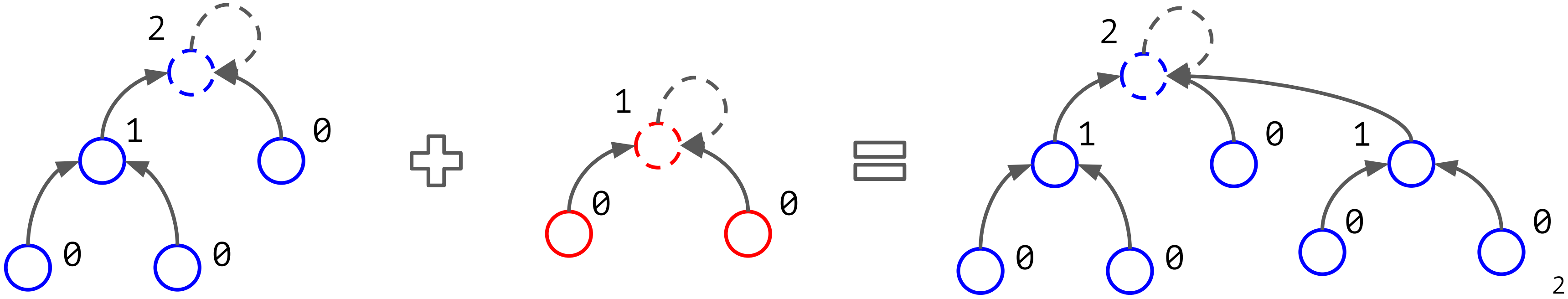


Union-find: объединение по рангу

$rank[x]$ - **высота поддерева** , растущего из x.

Это больше не так. А чем теперь является ранг?

Правило: при union, подвязываем дерево с меньшим рангом корня к дереву с большим



Union-find: сжатие путей

$rank[x]$ - максимальная высота поддерева, растущего из x.

Union-find: сжатие путей

$rank[x]$ - максимальная высота поддерева, растущего из x.

При этом:

1) Лемма о рангах остается в силе (т.е. все еще верно, что объектов с рангом r будет $\leq \frac{N}{2^r}$)

Union-find: сжатие путей

$rank[x]$ - максимальная высота поддерева, растущего из x.

При этом:

1) Лемма о рангах остается в силе (т.е. все еще верно, что объектов с рангом r будет $\leq \frac{N}{2^r}$)

2) Верно, что $rank[parent(x)] > rank[x]$ (это было верно и раньше, осталось и сейчас)

Union-find: сжатие путей

Теорема Хопкрофта-Ульмана: если реализована union-find, с оптимизациями "объединение по рангам" и "сжатие путей", то временная сложность последовательности из m операций над N элементами составляет $O(m * log^* N)$

Union-find: сжатие путей

Теорема Хопкрофта-Ульмана: если реализована union-find, с оптимизациями "объединение по рангам" и "сжатие путей", то временная сложность последовательности из m операций над N элементами составляет $O(m * log^* N)$

Здесь $log^* N$ - это итерированный логарифм (сколько раз надо применить к N логарифм, чтобы получилось значение ⩽ 1)

Union-find: сжатие путей

Теорема Хопкрофта-Ульмана: если реализована union-find, с оптимизациями "объединение по рангам" и "сжатие путей", то временная сложность последовательности из m операций над N элементами составляет $O(m * log^* N)$

Здесь $log^* N$ - это итерированный логарифм (сколько раз надо применить к N логарифм, чтобы получилось значение ⩽ 1)

$$\log^* n := \begin{cases} 0 & \text{if } n \le 1; \\ 1 + \log^*(\log n) & \text{if } n > 1 \end{cases}$$

Union-find: сжатие путей

Теорема Хопкрофта-Ульмана: если реализована union-find, с оптимизациями "объединение по рангам" и "сжатие путей", то временная сложность последовательности из m операций над N элементами составляет $O(m * log^* N)$

Здесь $log^* N$ - это итерированный логарифм (сколько раз надо применить к N логарифм, чтобы получилось значение ⩽ 1)

$$\log^* n := \begin{cases} 0 & \text{if } n \le 1; \\ 1 + \log^*(\log n) & \text{if } n > 1 \end{cases}$$

Чему, например, равно $log^*(2^{65536})$?

Union-find: сжатие путей

Теорема Хопкрофта-Ульмана: если реализована union-find, с оптимизациями "объединение по рангам" и "сжатие путей", то временная сложность последовательности из m операций над N элементами составляет $O(m * log^* N)$

Здесь $log^* N$ - это итерированный логарифм (сколько раз надо применить к N логарифм, чтобы получилось значение ⩽ 1)

$$\log^* n := \begin{cases} 0 & \text{if } n \le 1; \\ 1 + \log^*(\log n) & \text{if } n > 1 \end{cases}$$

Чему, например, равно $log^*(2^{65536}) = 5$

Union-find: сжатие путей

Теорема Хопкрофта-Ульмана: если реализована union-find, с оптимизациями "объединение по рангам" и "сжатие путей", то временная сложность последовательности из m операций над N элементами составляет $O(m * log^* N)$

Здесь $log^* N$ - это итерированный логарифм (сколько раз надо применить к N логарифм, чтобы получилось значение ⩽ 1)

$$\log^* n := \begin{cases} 0 & \text{if } n \le 1; \\ 1 + \log^*(\log n) & \text{if } n > 1 \end{cases}$$

Т.е. эта функция растет ОЧЕНЬ медленно.

Чему, например, равно $log^*(2^{65536}) = 5$

На практике это означает, что мы имеем фактически линейную сложность.

Теорема Хопкрофта-Ульмана: ... $O(m * log^* N)$

Допущение: будем говорить о случае, когда $m = \Omega(N)$, т.е. количество запросов сравнимо с количеством элементов.

Теорема Хопкрофта-Ульмана: … $O(m * log^* N)$

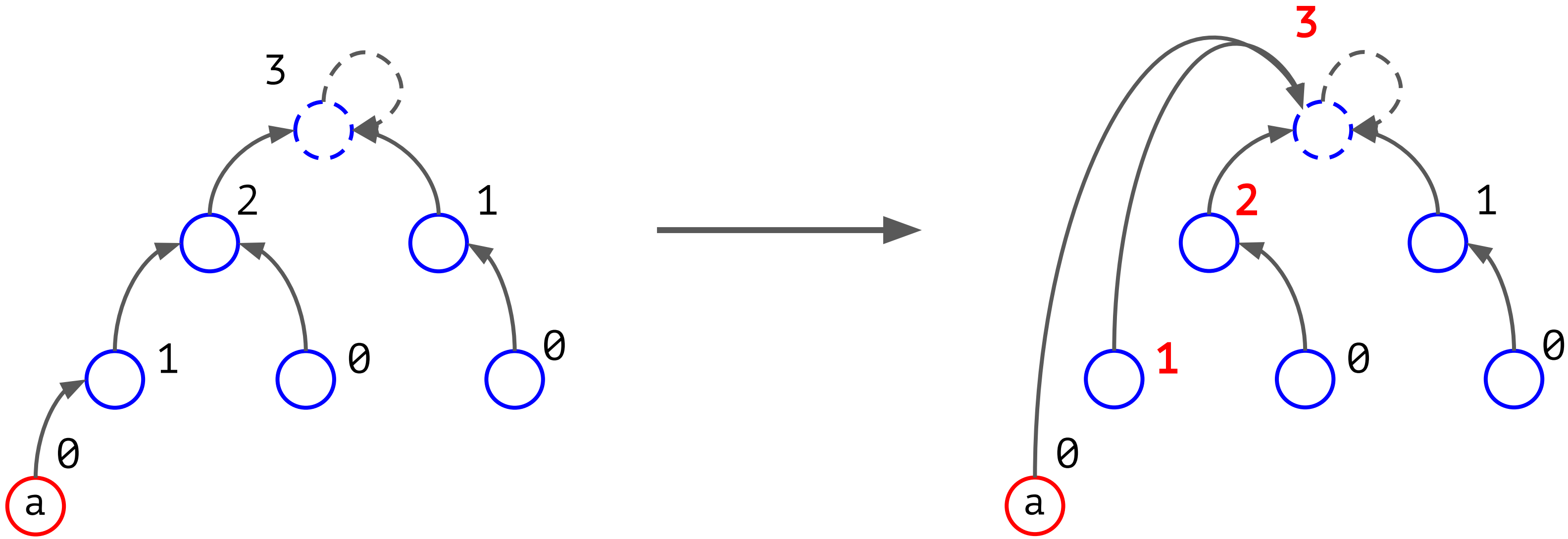
Допущение: будем говорить о случае, когда $m = \Omega(N)$, т.е. количество запросов сравнимо с количеством элементов.

Основная идея: следить, насколько сжатие путей ускоряет операции.

Теорема Хопкрофта-Ульмана: … $O(m * log^* N)$

Допущение: будем говорить о случае, когда $m = \Omega(N)$, т.е. количество запросов сравнимо с количеством элементов.

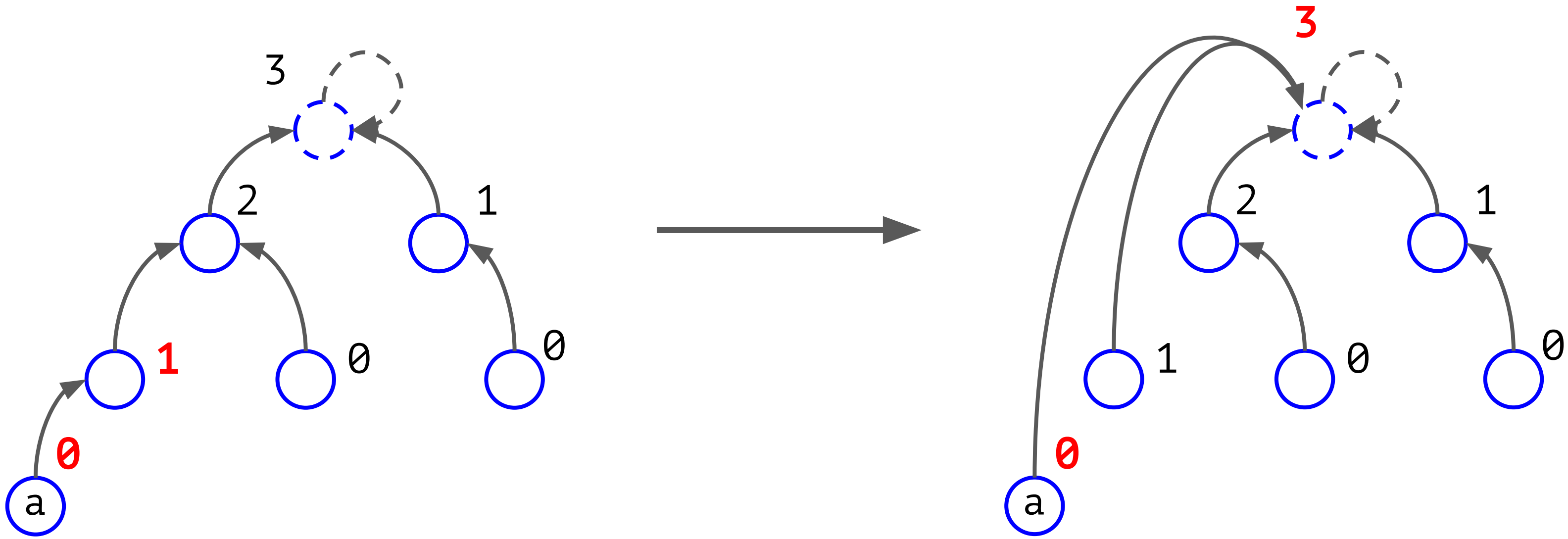
Основная идея: следить, насколько сжатие путей ускоряет операции.



Теорема Хопкрофта-Ульмана: … $O(m * log^* N)$

Допущение: будем говорить о случае, когда $m = \Omega(N)$, т.е. количество запросов сравнимо с количеством элементов.

Основная идея: следить, насколько сжатие путей ускоряет операции. Чем дальше наш ранг от родительского, тем сильнее "ускорились".



Теорема Хопкрофта-Ульмана: … $O(m * log^* N)$

Доказательство: разобьем все возможные ранги элементов на блоки по следующему правилу:

{0}, {1}, {2, 3, 4},

B1 B2 B3

Теорема Хопкрофта-Ульмана: … $O(m * log^* N)$

Доказательство: разобьем все возможные ранги элементов на блоки по следующему правилу:

{0}, {1}, {2, 3, 4}, {5, 6, …, 2^4}

B1 B2 B3 B4

Теорема Хопкрофта-Ульмана: … $O(m * log^* N)$

Доказательство: разобьем все возможные ранги элементов на блоки по следующему правилу:

$\{0\}, \{1\}, \{2, 3, 4\}, \{5, 6, \ldots, 2^4\}, \{17, 18, \ldots, 2^{2^4}\}$

B1 B2 B3 B4 B5

Теорема Хопкрофта-Ульмана: … $O(m * log^* N)$

Доказательство: разобьем все возможные ранги элементов на блоки по следующему правилу:

{0}, {1}, {2, 3, 4}, {5, 6, …, 2^4}, {17, 18, …, 2^{2^4}} … {…, N}

B1 B2 B3 B4 B5 B…

Теорема Хопкрофта-Ульмана: … $O(m * log^* N)$

Доказательство: разобьем все возможные ранги элементов на блоки по следующему правилу:

{0}, {1}, {2, 3, 4}, {5, 6, …, 2^4}, {17, 18, …, 2^{2^4}} … {…, N}

B1 B2 B3 B4 B5 B…

Сколько будет таких блоков? Сколько раз можно делать $2^{2^{2^{\cdots}}}$?

Теорема Хопкрофта-Ульмана: … $O(m * log^* N)$

Доказательство: разобьем все возможные ранги элементов на блоки по следующему правилу:

$\{0\}$, $\{1\}$, $\{2, 3, 4\}$, $\{5, 6, \ldots, 2^4\}$, $\{17, 18, \ldots, 2^{2^4}\}$ … $\{\ldots, N\}$

B1 B2 B3 B4 B5 B…

Сколько будет таких блоков? Сколько раз можно делать $2^{2^{2^{\cdots}}}$?
Как раз $log^* N$. С учетом первых блоков: $O(log^* N)$

Теорема Хопкрофта-Ульмана: … $O(m * log^* N)$

Доказательство: разобьем все возможные ранги элементов на блоки по следующему правилу:

{0}, {1}, {2, 3, 4}, {5, 6, …, 2^4}, {17, 18, …, 2^{2^4}} … {…, N}

B1 B2 B3 B4 B5 B…

Сколько будет таких блоков? Сколько раз можно делать $2^{2^{2^{\cdots}}}$? Как раз $log^* N$. С учетом первых блоков: $O(log^* N)$

Ключевая идея: дальше будем считать, что прогресс сжатия пути хороший, если после него ранг элемента и его предка находятся в разных блоках рангов.

Теорема Хопкрофта-Ульмана: … $O(m * log^* N)$

Доказательство: разобьем все возможные ранги элементов на блоки по следующему правилу:

{0}, {1}, {2, 3, 4}, {5, 6, …, 2^4}, {17, 18, …, 2^{2^4}} … {…, N}

B1 B2 B3 B4 B5 B…

Назовем элемент x в UF хорошим, если верно одно из двух:

1. x - корень, либо parent(x) - корень

Теорема Хопкрофта-Ульмана: … $O(m * log^* N)$

Доказательство: разобьем все возможные ранги элементов на блоки по следующему правилу:

$\{0\}$, $\{1\}$, $\{2, 3, 4\}$, $\{5, 6, \ldots, 2^4\}$, $\{17, 18, \ldots, 2^{2^4}\}$ … $\{\ldots, N\}$

B1 B2 B3 B4 B5 B…

Назовем элемент x в UF хорошим, если верно одно из двух:

1. x - корень, либо parent(x) - корень
2. rank[parent(x)] находится в большем блоке рангов, чем rank[x]

Теорема Хопкрофта-Ульмана: … $O(m * log^* N)$

Доказательство: разобьем все возможные ранги элементов на блоки по следующему правилу:

$\{0\}$, $\{1\}$, $\{2, 3, 4\}$, $\{5, 6, …, 2^4\}$, $\{17, 18, …, 2^{2^4}\}$ … $\{…, N\}$

B1 B2 B3 B4 B5 B…

Назовем элемент x в UF хорошим, если верно одно из двух:

1. x - корень, либо parent(x) - корень
2. rank[parent(x)] находится в большем блоке рангов, чем rank[x]

Иначе - элемент плохой.

Теорема Хопкрофта-Ульмана: … $O(m * log^* N)$

Доказательство: …

Вопрос: сколько "хороших" элементов мы можем встретить в худшем случае во время работы операции find?

Теорема Хопкрофта-Ульмана: … $O(m * log^* N)$

Доказательство: …

Вопрос: сколько "хороших" элементов мы можем встретить в худшем случае во время работы операции find?

Ответ: мы точно встретим корень и его прямого потомка (это уже два), а в остальном - не больше, чем количество блоков (ведь наши предки каждый раз будут в следующем блоке).

Теорема Хопкрофта-Ульмана: … $O(m * log^* N)$

Доказательство: …

Вопрос: сколько "хороших" элементов мы можем встретить в худшем случае во время работы операции find?

Ответ: мы точно встретим корень и его прямого потомка (это уже два), а в остальном - не больше, чем количество блоков (ведь наши предки каждый раз будут в следующем блоке). Т.е. всего: $O(log^* N)$

Теорема Хопкрофта-Ульмана: … $O(m * log^* N)$

Доказательство: …

Вопрос: сколько "хороших" элементов мы можем встретить в худшем случае во время работы операции find?

Ответ: мы точно встретим корень и его прямого потомка (это уже два), а в остальном - не больше, чем количество блоков (ведь наши предки каждый раз будут в следующем блоке). Т.е. всего: $O(log^* N)$

Теперь оценим общее количество работы за **m** операций.

Теорема Хопкрофта-Ульмана: ... $O(m * log^* N)$

Доказательство: ...

Вопрос: сколько "хороших" элементов мы можем встретить в худшем случае во время работы операции find?

Ответ: мы точно встретим корень и его прямого потомка (это уже два), а в остальном - не больше, чем количество блоков (ведь наши предки каждый раз будут в следующем блоке). Т.е. всего: $O(log^* N)$

Теперь оценим общее количество работы за **m** операций. Общее количество работы = посещение хороших элементов + посещение плохих.

Теорема Хопкрофта-Ульмана: … $O(m * log^* N)$

Доказательство: …

Вопрос: сколько "хороших" элементов мы можем встретить в худшем случае во время работы операции find?

Ответ: мы точно встретим корень и его прямого потомка (это уже два), а в остальном - не больше, чем количество блоков (ведь наши предки каждый раз будут в следующем блоке). Т.е. всего: $O(log^* N)$

Теперь оценим общее количество работы за **m** операций. Общее количество работы = $\underbrace{\text{посещение хороших элементов}}_{O(m * log^* N)}$ + $\underbrace{\text{посещение плохих}}_{???}$.

Теорема Хопкрофта-Ульмана: … $O(m * log^* N)$

Доказательство: …

Теперь оценим общее количество работы за **m** операций. Общее количество работы = $\underbrace{\text{посещение хороших элементов}}_{O(m * log^* N)}$ + $\underbrace{\text{посещение плохих}}_{???}$.

Рассмотрим конкретный блок рангов: $\{k+1, k+2, \ldots, 2^k\}$

Теорема Хопкрофта-Ульмана: … $O(m * log^* N)$

Доказательство: …

Теперь оценим общее количество работы за **m** операций. Общее количество работы = $\underbrace{\text{посещение хороших элементов}}_{O(m * log^* N)}$ + $\underbrace{\text{посещение плохих}}_{???}$.

Рассмотрим конкретный блок рангов: $\{k+1, k+2, \ldots, 2^k\}$

Заметим, что каждый раз, когда мы посещаем элемент, мы обновляем его parent-a, увеличивая разрыв между рангами.

Теорема Хопкрофта-Ульмана: … $O(m * log^* N)$

Доказательство: …

Теперь оценим общее количество работы за **m** операций. Общее количество работы = $\underbrace{\text{посещение хороших элементов}}_{O(m * log^* N)}$ + $\underbrace{\text{посещение плохих}}_{???}$.

Рассмотрим конкретный блок рангов: $\{k+1, k+2, \ldots, 2^k\}$

Заметим, что каждый раз, когда мы посещаем элемент, мы обновляем его parent-а, увеличивая разрыв между рангами.

Пусть мы посещаем "плохой" элемент с рангом из этого блока.

Теорема Хопкрофта-Ульмана: … $O(m * log^* N)$

Доказательство: …

Теперь оценим общее количество работы за **m** операций. Общее количество работы = $\underbrace{\text{посещение хороших элементов}}_{O(m * log^* N)}$ + $\underbrace{\text{посещение плохих}}_{???}$.

Рассмотрим конкретный блок рангов: $\{k+1, k+2, \ldots, 2^k\}$

Заметим, что каждый раз, когда мы посещаем элемент, мы обновляем его parent-а, увеличивая разрыв между рангами.

Пусть мы посещаем "плохой" элемент с рангом из этого блока. Сколько раз можно его посетить, перед тем, как он станет "хорошим" (т.е. его parent будет в след. блоке)?

Теорема Хопкрофта-Ульмана: … $O(m * log^* N)$

Доказательство: разобьем все возможные ранги элементов на блоки по следующему правилу:

$\{0\}, \{1\}, \{2, 3, 4\}, \{5, 6, \ldots, 2^4\}, \{17, 18, \ldots, 2^{2^4}\} \ldots \{\ldots, N\}$

B1 B2 B3 B4 B5 B…

Назовем элемент x в UF хорошим, если верно одно из двух:

1. x - корень, либо parent(x) - корень
2. rank[parent(x)] находится в большем блоке рангов, чем rank[x]

Иначе - элемент плохой.

Теорема Хопкрофта-Ульмана: … $O(m * log^* N)$

Доказательство: …

Теперь оценим общее количество работы за **m** операций. Общее количество работы = $\underbrace{\text{посещение хороших элементов}}_{O(m * log^* N)}$ + $\underbrace{\text{посещение плохих}}_{???}$.

Рассмотрим конкретный блок рангов: $\{k+1, k+2, \ldots, 2^k\}$

Заметим, что каждый раз, когда мы посещаем элемент, мы обновляем его parent-а, увеличивая разрыв между рангами.

Пусть мы посещаем "плохой" элемент с рангом из этого блока. Сколько раз можно его посетить, перед тем, как он станет "хорошим" (т.е. его parent будет в след. блоке)? $\leq 2^k$

Теорема Хопкрофта-Ульмана: … $O(m * log^* N)$

Доказательство: …

Теперь оценим общее количество работы за **m** операций. Общее количество работы = $\underbrace{\text{посещение хороших элементов}}_{O(m * log^* N)}$ + $\underbrace{\text{посещение плохих}}_{???}$.

Рассмотрим конкретный блок рангов: $\{k+1, k+2, \ldots, 2^k\}$

Сколько раз можно посетить "плохой" элемент, перед тем, как он станет "хорошим" (его parent будет в след. блоке)? $\leq 2^k$

Теорема Хопкрофта-Ульмана: … $O(m * log^* N)$

…

Теперь оценим общее количество работы за **m** операций. Общее количество работы = $\underbrace{\text{посещение хороших элементов}}_{O(m * log^* N)}$ + $\underbrace{\text{посещение плохих}}_{???}$.

Рассмотрим конкретный блок рангов: $\{k+1, k+2, \ldots, 2^k\}$

Сколько раз можно посетить "плохой" элемент, перед тем, как он станет "хорошим" (его parent будет в след. блоке)? $\leq 2^k$

А сколько всего "плохих" элементов с рангами из блока?

Теорема Хопкрофта-Ульмана: … $O(m * log^* N)$

…

Теперь оценим общее количество работы за **m** операций. Общее количество работы = $\underbrace{\text{посещение хороших элементов}}_{O(m * log^* N)}$ + $\underbrace{\text{посещение плохих}}_{???}$.

Рассмотрим конкретный блок рангов: $\{k+1, k+2, \ldots, 2^k\}$

Сколько раз можно посетить "плохой" элемент, перед тем, как он станет "хорошим" (его parent будет в след. блоке)? $\leq 2^k$

А сколько всего "плохих" элементов с рангами из блока? Уж точно не больше, всех элементов с соответствующими рангами.

Теорема Хопкрофта-Ульмана: … $O(m * log^* N)$

…

Теперь оценим общее количество работы за **m** операций. Общее количество работы = $\underbrace{\text{посещение хороших элементов}}_{O(m * log^* N)}$ + $\underbrace{\text{посещение плохих}}_{???}$.

Рассмотрим конкретный блок рангов: $\{k+1, k+2, \ldots, 2^k\}$

Сколько раз можно посетить "плохой" элемент, перед тем, как он станет "хорошим" (его parent будет в след. блоке)? $\leq 2^k$

А сколько всего "плохих" элементов с рангами из блока? Уж точно не больше, всех элементов с соответствующими рангами. Используем лемму о рангах!

Union-find: объединение по рангу (анализ)

Лемма о рангах: пусть в union-find есть N элементов, тогда для любого $r \geq 0$ верно, что объектов с рангом r не больше, чем $\frac{N}{2^r}$.

Теорема Хопкрофта-Ульмана: … $O(m * log^* N)$

…

Теперь оценим общее количество работы за **m** операций. Общее количество работы = $\underbrace{\text{посещение хороших элементов}}_{O(m * log^* N)}$ + $\underbrace{\text{посещение плохих}}_{???}$.

Рассмотрим конкретный блок рангов: $\{k+1, k+2, \ldots, 2^k\}$

Сколько раз можно посетить "плохой" элемент, перед тем, как он станет "хорошим" (его parent будет в след. блоке)? $\leq 2^k$

"Плохих" элементов с рангами из блока $\leq \sum_{i=k+1}^{2^k}$

Теорема Хопкрофта-Ульмана: … $O(m * log^* N)$

…

Теперь оценим общее количество работы за **m** операций. Общее количество работы = $\underbrace{\text{посещение хороших элементов}}_{O(m * log^* N)}$ + $\underbrace{\text{посещение плохих}}_{???}$.

Рассмотрим конкретный блок рангов: $\{k+1, k+2, \ldots, 2^k\}$

Сколько раз можно посетить "плохой" элемент, перед тем, как он станет "хорошим" (его parent будет в след. блоке)? $\leq 2^k$

"Плохих" элементов с рангами из блока $\leq \sum_{i=k+1}^{2^k} \frac{N}{2^i}$

Теорема Хопкрофта-Ульмана: … $O(m * log^* N)$

…

Теперь оценим общее количество работы за **m** операций. Общее количество работы = $\underbrace{\text{посещение хороших элементов}}_{O(m * log^* N)}$ + $\underbrace{\text{посещение плохих}}_{???}$.

Рассмотрим конкретный блок рангов: $\{k+1, k+2, \ldots, 2^k\}$

Сколько раз можно посетить "плохой" элемент, перед тем, как он станет "хорошим" (его parent будет в след. блоке)? $\leq 2^k$

"Плохих" элементов с рангами из блока $\leq \sum_{i=k+1}^{2^k} \frac{N}{2^i} \leq \frac{N}{2^k}$

Теорема Хопкрофта-Ульмана: … $O(m * log^* N)$

…

Теперь оценим общее количество работы за **m** операций. Общее количество работы = $\underbrace{\text{посещение хороших элементов}}_{O(m * log^* N)}$ + $\underbrace{\text{посещение плохих}}_{???}$.

Рассмотрим конкретный блок рангов: $\{k+1, k+2, \ldots, 2^k\}$

Сколько раз можно посетить "плохой" элемент, перед тем, как он станет "хорошим" (его parent будет в след. блоке)? $\leq \boxed{2^k}$

"Плохих" элементов с рангами из блока $\leq \sum_{i=k+1}^{2^k} \frac{N}{2^i} \leq \boxed{\frac{N}{2^k}}$

Теорема Хопкрофта-Ульмана: … $O(m * log^* N)$

…

Теперь оценим общее количество работы за **m** операций. Общее количество работы = $\underbrace{\text{посещение хороших элементов}}_{O(m * log^* N)}$ + $\underbrace{\text{посещение плохих}}_{???}$.

Рассмотрим конкретный блок рангов: $\{k+1, k+2, \ldots, 2^k\}$

Сколько раз можно посетить "плохой" элемент, перед тем, как он станет "хорошим" (его parent будет в след. блоке)? $\leq \boxed{2^k}$

"Плохих" элементов с рангами из блока $\leq \sum_{i=k+1}^{2^k} \frac{N}{2^i} \leq \boxed{\frac{N}{2^k}}$

Тогда посещений "плохих" элементов для каждого блока $\leq N$

Теорема Хопкрофта-Ульмана: … $O(m * log^* N)$

…

Теперь оценим общее количество работы за **m** операций. Общее количество работы = $\underbrace{\text{посещение хороших элементов}}_{O(m * log^* N)}$ + $\underbrace{\text{посещение плохих}}_{???}$.

Рассмотрим конкретный блок рангов: $\{k+1, k+2, \ldots, 2^k\}$

Сколько раз можно посетить "плохой" элемент, перед тем, как он станет "хорошим" (его parent будет в след. блоке)? $\leq \boxed{2^k}$

"Плохих" элементов с рангами из блока $\leq \sum_{i=k+1}^{2^k} \frac{N}{2^i} \leq \boxed{\frac{N}{2^k}}$

Тогда посещений "плохих" элементов для каждого блока $\leq N$
Всего блоков: $O(log^* N)$

Теорема Хопкрофта-Ульмана: … $O(m * log^* N)$

…

Теперь оценим общее количество работы за **m** операций. Общее количество работы = $\underbrace{\text{посещение хороших элементов}}_{O(m * log^* N)}$ + $\underbrace{\text{посещение плохих}}_{O(N * log^* N)}$.

Рассмотрим конкретный блок рангов: $\{k+1, k+2, \ldots, 2^k\}$

Сколько раз можно посетить "плохой" элемент, перед тем, как он станет "хорошим" (его parent будет в след. блоке)? $\le \boxed{2^k}$

"Плохих" элементов с рангами из блока $\le \sum_{i=k+1}^{2^k} \frac{N}{2^i} \le \boxed{\frac{N}{2^k}}$

Тогда посещений "плохих" элементов для каждого блока $\le N$

Всего блоков: $O(log^* N)$ => на плохие элементы тратим $O(N * log^* N)$

Теорема Хопкрофта-Ульмана: … $O(m * log^* N)$

…

Теперь оценим общее количество работы за **m** операций. Общее количество работы = $\underbrace{\text{посещение хороших элементов}}_{O(m * log^* N)}$ + $\underbrace{\text{посещение плохих}}_{O(N * log^* N)}$.

Тогда общее количество работы: $O((m + N) * log^* N)$

Теорема Хопкрофта-Ульмана: … $O(m * log^* N)$

…

Теперь оценим общее количество работы за **m** операций. Общее количество работы = $\underbrace{\text{посещение хороших элементов}}_{O(m * log^* N)}$ + $\underbrace{\text{посещение плохих}}_{O(N * log^* N)}$.

Тогда общее количество работы: $O((m + N) * log^* N)$

Рассматриваем случай, когда $m = \Omega(N)$ => ответ: $O(m * log^* N)$ $\square$

Теорема Хопкрофта-Ульмана: … $O(m * log^* N)$

…

Теперь оценим общее количество работы за **m** операций. Общее количество работы = $\underbrace{\text{посещение хороших элементов}}_{O(m * log^* N)}$ + $\underbrace{\text{посещение плохих}}_{O(N * log^* N)}$.

Тогда общее количество работы: $O((m + N) * log^* N)$

Рассматриваем случай, когда $m = \Omega(N)$ => ответ: $O(m * log^* N)$ $\square$

(для случая, когда запросов мало рассматривается каждое дерево отдельно)

Union-find: сжатие путей

Теорема Хопкрофта-Ульмана: если реализована union-find, с оптимизациями "объединение по рангам" и "сжатие путей", то временная сложность последовательности из m операций над N элементами составляет $O(m * log^* N)$

Здесь $log^* N$ - это итерированный логарифм (сколько раз надо применить к N логарифм, чтобы получилось значение ⩽ 1)

$$\log^* n := \begin{cases} 0 & \text{if } n \le 1; \\ 1 + \log^*(\log n) & \text{if } n > 1 \end{cases}$$

Т.е. эта функция растет ОЧЕНЬ медленно.

Чему, например, равно $log^*(2^{65536}) = 5$

На практике это означает, что мы имеем фактически линейную сложность.

Union-find: сжатие путей

Теорема Хопкрофта-Ульмана: если реализована union-find, с оптимизациями "объединение по рангам" и "сжатие путей", то временная сложность последовательности из m операций над N элементами составляет $O(m * log^* N)$

Но и это еще не все!
На самом деле все еще быстрее.

Union-find: сжатие путей

Теорема Тарьяна: если реализована union-find, с оптимизациями "объединение по рангам" и "сжатие путей", то временная сложность последовательности из m операций над N элементами составляет $O(m * \alpha(N))$, где $\alpha(N)$ - обратная функция Аккермана.

Функция Аккермана

Пусть $k \geq 0; r \geq 1$. Тогда определим $A_k(r)$ рекурсивно.

Базовый случай: $A_0(r) = r + 1$

Функция Аккермана

Пусть $k \geq 0; r \geq 1$. Тогда определим $A_k(r)$ рекурсивно.

Базовый случай: $A_0(r) = r + 1$

Общий случай: $A_k(r) =$ применение A_{k-1} r раз к $r =$

Функция Аккермана

Пусть $k \geq 0; r \geq 1$. Тогда определим $A_k(r)$ рекурсивно.

Базовый случай: $A_0(r) = r + 1$

Общий случай: $A_k(r) =$ применение A_{k-1} r раз к $r =$

$$= (\underbrace{A_{k-1} \circ A_{k-1} \circ \cdots \circ A_{k-1}}_{r})(r)$$

Функция Аккермана

Пусть $k \geq 0; r \geq 1$. Тогда определим $A_k(r)$ рекурсивно.

Базовый случай: $A_0(r) = r + 1$

Общий случай: $A_k(r) =$ применение A_{k-1} r раз к $r =$

$$= (\underbrace{A_{k-1} \circ A_{k-1} \circ \cdots \circ A_{k-1}}_{r})(r)$$

Пример: $A_1(r) = ?$

Функция Аккермана

Пусть $k \geq 0; r \geq 1$. Тогда определим $A_k(r)$ рекурсивно.

Базовый случай: $A_0(r) = r + 1$

Общий случай: $A_k(r) =$ применение A_{k-1} r раз к $r =$

$$= (\underbrace{A_{k-1} \circ A_{k-1} \circ \cdots \circ A_{k-1}}_{r})(r)$$

Пример: $A_1(r) = 2r$

Функция Аккермана

Пусть $k \geq 0; r \geq 1$. Тогда определим $A_k(r)$ рекурсивно.

Базовый случай: $A_0(r) = r + 1$

Общий случай: $A_k(r) =$ применение A_{k-1} r раз к $r =$

$$= (\underbrace{A_{k-1} \circ A_{k-1} \circ \cdots \circ A_{k-1}}_{r})(r)$$

Пример: $A_1(r) = 2r$

$A_2(r) =$

Функция Аккермана

Пусть $k \geq 0; r \geq 1$. Тогда определим $A_k(r)$ рекурсивно.

Базовый случай: $A_0(r) = r + 1$

Общий случай: $A_k(r) =$ применение A_{k-1} r раз к $r =$

$$= (\underbrace{A_{k-1} \circ A_{k-1} \circ \cdots \circ A_{k-1}}_{r})(r)$$

Пример: $A_1(r) = 2r$

$$A_2(r) = r * 2^r$$

Функция Аккермана

Пусть $k \geq 0; r \geq 1$. Тогда определим $A_k(r)$ рекурсивно.

Базовый случай: $A_0(r) = r + 1$

Общий случай: $A_k(r) =$ применение A_{k-1} r раз к r $=$

$$= (\underbrace{A_{k-1} \circ A_{k-1} \circ \cdots \circ A_{k-1}}_{r})(r)$$

Пример: $A_1(r) = 2r$ $\qquad$ $A_3(2) =$

$A_2(r) = r * 2^r$

Функция Аккермана

Пусть $k \geq 0; r \geq 1$. Тогда определим $A_k(r)$ рекурсивно.

Базовый случай: $A_0(r) = r + 1$

Общий случай: $A_k(r) =$ применение A_{k-1} r раз к $r =$

$$= (\underbrace{A_{k-1} \circ A_{k-1} \circ \cdots \circ A_{k-1}}_{r})(r)$$

Пример: $A_1(r) = 2r$ $\qquad A_3(2) = A_2(A_2(2)) =$

$A_2(r) = r * 2^r$

Функция Аккермана

Пусть $k \geq 0; r \geq 1$. Тогда определим $A_k(r)$ рекурсивно.

Базовый случай: $A_0(r) = r + 1$

Общий случай: $A_k(r) =$ применение A_{k-1} r раз к $r =$

$$= (\underbrace{A_{k-1} \circ A_{k-1} \circ \cdots \circ A_{k-1}}_{r})(r)$$

Пример: $A_1(r) = 2r$

$A_2(r) = r * 2^r$

$A_3(2) = A_2(A_2(2)) = A_2(8) =$

Функция Аккермана

Пусть $k \geq 0; r \geq 1$. Тогда определим $A_k(r)$ рекурсивно.

Базовый случай: $A_0(r) = r + 1$

Общий случай: $A_k(r) =$ применение A_{k-1} r раз к $r =$

$$= (\underbrace{A_{k-1} \circ A_{k-1} \circ \cdots \circ A_{k-1}}_{r})(r)$$

Пример: $A_1(r) = 2r$ $\qquad A_3(2) = A_2(A_2(2)) = A_2(8) = 8 * 2^8 = 2048$

$A_2(r) = r * 2^r$

Функция Аккермана

Пусть $k \geq 0; r \geq 1$. Тогда определим $A_k(r)$ рекурсивно.

Базовый случай: $A_0(r) = r + 1$

Общий случай: $A_k(r) =$ применение A_{k-1} r раз к $r =$

$$= (\underbrace{A_{k-1} \circ A_{k-1} \circ \cdots \circ A_{k-1}}_{r})(r)$$

Пример: $A_1(r) = 2r$

$A_2(r) = r * 2^r$

$$A_3(2) = A_2(A_2(2)) = A_2(8) = 8 * 2^8 = 2048$$

Чем больше индекс, тем более взрывной рост у функции Аккермана

Обратная функция Аккермана

Пусть $n \geq 4$, определим обратную функцию Аккермана, как:

$\alpha(n) =$ минимальный k такой, что: $A_k(2) \geq n$

Обратная функция Аккермана

Пусть $n \geq 4$, определим обратную функцию Аккермана, как:

$\alpha(n) =$ минимальный k такой, что: $A_k(2) \geq n$

Примеры:

$$\alpha(n) = 1, \text{ если } n = 4$$

Обратная функция Аккермана

Пусть $n \geq 4$, определим обратную функцию Аккермана, как:

$\alpha(n) =$ минимальный k такой, что: $A_k(2) \geq n$

Примеры:

$$\alpha(n) = 1, \text{ если } n = 4$$

$$\alpha(n) = 2, \text{ если } n \in \{5, 6, 7, 8\}$$

$$\alpha(n) = 3, \text{ если } n \in \{9, 10, \ldots 2048\}$$

Обратная функция Аккермана

Пусть $n \geq 4$, определим обратную функцию Аккермана, как:

$\alpha(n) =$ минимальный k такой, что: $A_k(2) \geq n$

Примеры:

$$\alpha(n) = 1, \text{ если } n = 4$$

$$\alpha(n) = 2, \text{ если } n \in \{5, 6, 7, 8\}$$

$$\alpha(n) = 3, \text{ если } n \in \{9, 10, \ldots 2048\}$$

$$\alpha(n) = 4, \text{ если } n \in \{2048, \ldots, \xi\}$$

$$\xi = \left.2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}\right\} \text{высота } 2048$$

Обратная функция Аккермана

Пусть $n \geq 4$, определим обратную функцию Аккермана, как:

$\alpha(n) =$ минимальный k такой, что: $A_k(2) \geq n$

Примеры:

$\alpha(n) = 1,$ если $n = 4$

$\alpha(n) = 2,$ если $n \in \{5, 6, 7, 8\}$

$\alpha(n) = 3,$ если $n \in \{9, 10, \dots 2048\}$

$\alpha(n) = 4,$ если $n \in \{2048, \dots, \xi\}$

$$\xi = \left.2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}\right\}\text{высота 2048}$$

Сравним:

$log^*(n) = 1,$ если $n = 2$

$log^*(n) = 2,$ если $n \in \{3, 4\}$

$log^*(n) = 3,$ если $n \in \{5, \dots, 16\}$

$log^*(n) = 4,$ если $n \in \{17, \dots, 65536\}$

Обратная функция Аккермана

Пусть $n \geq 4$, определим обратную функцию Аккермана, как:

$\alpha(n) =$ минимальный k такой, что: $A_k(2) \geq n$

Примеры:

$$\alpha(n) = 1, \text{ если } n = 4$$

$$\alpha(n) = 2, \text{ если } n \in \{5, 6, 7, 8\}$$

$$\alpha(n) = 3, \text{ если } n \in \{9, 10, \ldots 2048\}$$

$$\alpha(n) = 4, \text{ если } n \in \{2048, \ldots, \xi\}$$

$$\xi = \left. 2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}} \right\} \text{высота } 2048$$

Сравним:

$$log^*(n) = 1, \text{ если } n = 2$$

$$log^*(n) = 2, \text{ если } n \in \{3, 4\}$$

$$log^*(n) = 3, \text{ если } n \in \{5, \ldots, 16\}$$

$$log^*(n) = 4, \text{ если } n \in \{17, \ldots, 65536\}$$

Обратная функция Аккермана
намного медленнее растет!

Union-find: сжатие путей

Теорема Тарьяна: если реализована union-find, с оптимизациями "объединение по рангам" и "сжатие путей", то временная сложность последовательности из m операций над N элементами составляет $O(m * \alpha(N))$, где $\alpha(N)$ - обратная функция Аккермана.

На практике можно считать, что $\alpha(N)$ не превосходит 4, т.е. получили еще более близкую к линейной сложность.

Union-find: сжатие путей

Теорема Тарьяна: если реализована union-find, с оптимизациями "объединение по рангам" и "сжатие путей", то временная сложность последовательности из m операций над N элементами составляет $O(m * \alpha(N))$, где $\alpha(N)$ - обратная функция Аккермана.

На практике можно считать, что $\alpha(N)$ не превосходит 4, т.е. получили еще более близкую к линейной сложность.

И быстрее и нельзя, эта оценка является асимптотически точной, т.е. имеем: $\Omega(m * \alpha(N))$

Мини-задача #35 (1 балл)

Пусть было дерево, в которое добавили одно ребро, сделав его графом. Ваша задача найти некоторое ребро, удалив которое вы снова получите дерево. Если таких ребер несколько - взять последнее из перечисленных во входных данных.

https://leetcode.com/problems/redundant-connection/

В union-find реализовать оптимизации: либо union by size, либо сжатие путей + объединением по рангам.

Мини-задача #35 (1-2 балла)

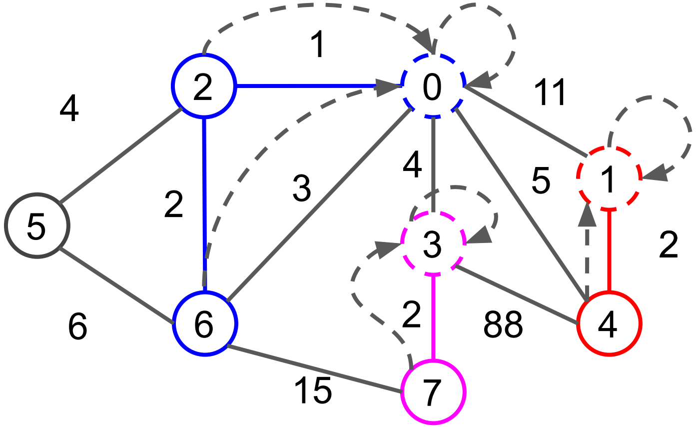
Пусть было дерево, в которое добавили одно ребро, сделав его графом. Ваша задача найти некоторое ребро, удалив которое вы снова получите дерево. Если таких ребер несколько - взять последнее из перечисленных во входных данных.

https://leetcode.com/problems/redundant-connection/

В union-find реализовать оптимизации: либо union by size, либо сжатие путей + объединением по рангам..

За 1 доп. балл решите задачу для ориентированного графа:
https://leetcode.com/problems/redundant-connection-ii

Алгоритм Краскала: union-find



Итого:

1) O(|E|*log(|V|)) за сортировку

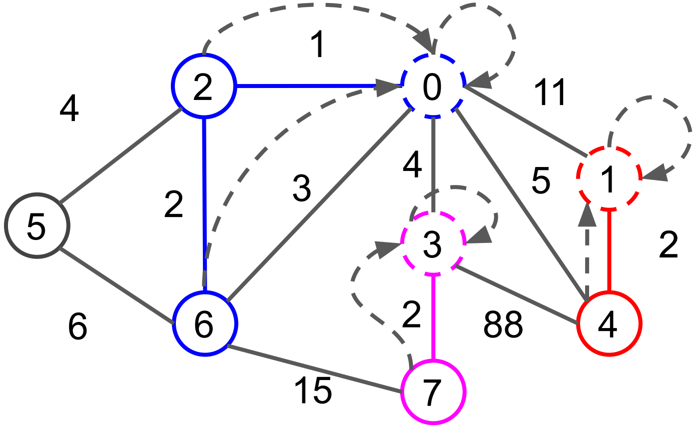
2) O(|E|) за проверку ребер на циклы

3) O(|V|*log(|V|)) за union-ы

Что здесь на самом деле произошло: хотя каждая операция union может занимать линейное время в худшем случае, т.е. дает честное O(|V|), сложность последовательности union-ов дает всего O(|V|*log(|V|))!

Как такое называется? Амортизационная сложность! (в этот раз без монет)

Алгоритм Краскала: union-find



Итого:

1) O(|E|*log(|V|)) за сортировку

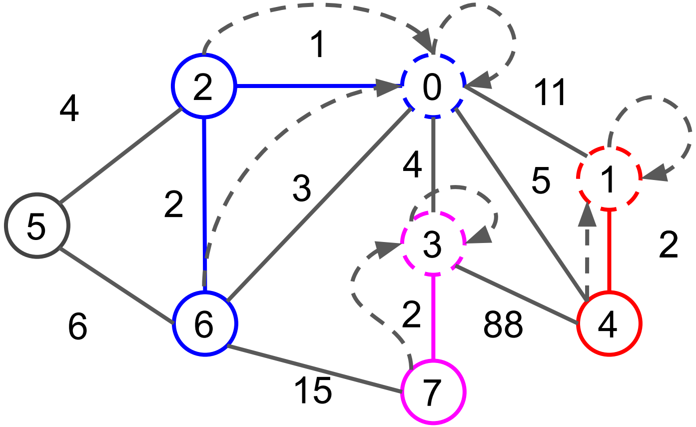
2) O(|E|) за проверку ребер на циклы

3) ~~O(|V|*log(|V|))~~ за union-ы
O(|E|*log(|V|)) из-за связности графа

Но можем ли мы еще лучше?

Догнали Приму!

Алгоритм Краскала: union-find



Итого:

1) O(|E|*log(|V|)) за сортировку

2) O(|E|) за проверку ребер на циклы

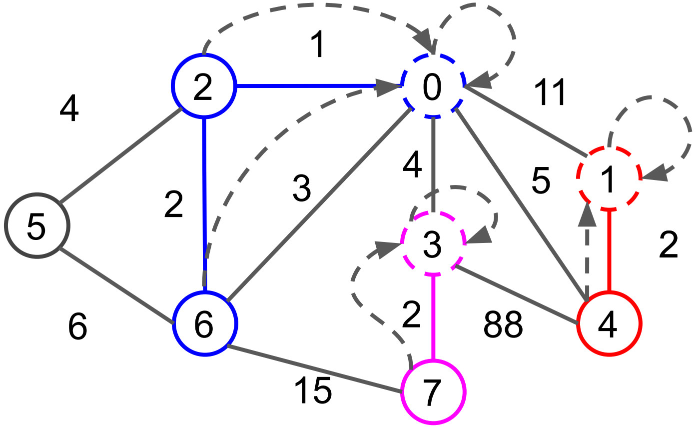
3) ~~O(|V|*log(|V|))~~ за union-ы
~~O(|E|*log(|V|))~~
O(|E|*a(|V|))

Но можем ли мы еще лучше?

Добавим оптимизации: получили намного лучшие константы, но...

Догнали Приму!

Алгоритм Краскала: union-find



Итого:

1) O(|E|*log(|V|)) за сортировку

2) O(|E|) за проверку ребер на циклы

3) ~~O(|V|*log(|V|))~~ за union-ы
~~O(|E|*log(|V|))~~
O(|E|*a(|V|))

Но можем ли мы еще лучше?

Догнали Приму!

Добавим оптимизации: получили
намного лучшие константы, но сортировка теперь мажорирует 😔

Другие алгоритмы поиска остовного дерева

Другие алгоритмы поиска остовного дерева

Алгоритм Каргера & Клейна & Тарьяна (1995): рандомизированный алгоритм, который дает $O(|E|)$ в среднем.

Другие алгоритмы поиска остовного дерева

Алгоритм Каргера & Клейна & Тарьяна (1995): рандомизированный алгоритм, который дает O(|E|) в среднем.

Алгоритм Чейзелла (2000): детерминированный (!!!) алгоритм, работающий за O(|E|*a(|V|)), где a - снова обратная функция Аккермана.

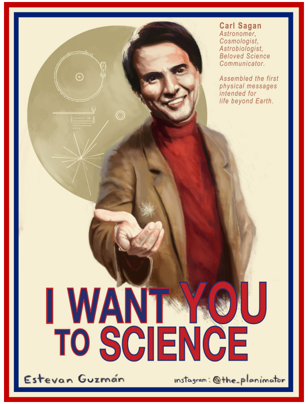
Другие алгоритмы поиска остовного дерева

Алгоритм Каргера & Клейна & Тарьяна (1995): рандомизированный алгоритм, который дает O(|E|) в среднем.

Алгоритм Чейзелла (2000): детерминированный (!!!) алгоритм, работающий за O(|E|*a(|V|)), где a - снова обратная функция Аккермана.

Существует ли линейный и детерминированный алгоритм за O(|E|)?

Мы не знаем. Возможно именно вы его придумаете!



Takeaways

- Жадные алгоритмы на графах - не только Дейкстра
- Разные подходы к жадности у Примы и Крускала
- Улучшение жадных алгоритмов через структуры данных

Takeaways

- Жадные алгоритмы на графах - не только Дейкстра

- Разные подходы к жадности у Примы и Крускала

- Улучшение жадных алгоритмов через структуры данных

- Union-find - простейшая в реализации структура данных с почти линейной сложностью 💜