

Руководство для пользователя:

- **Добавление вершины – ЛКМ**
- **Выделение вершины – ЛКМ**
- **Добавление ребра – выделение нужной вершины + ЛКМ по вершине, к которой хотите добавить ребро**
- **Удаление ребра - выделение нужной вершины + ПКМ по вершине, у которой хотите удалить ребро**
- **Удаление вершины – SHIFT + ЛКМ**
- **Сменить вес - выделение нужной вершины + ЛКМ по вершине, к которой хотите поменять вес**

При графическом задании алгоритмы доступны только при добавлении хотя бы одной вершины!!!

Компоненты сильной связности

Компоненты сильной связности - максимальные сильно связанные подграфы

Инвертирование графа - смена направлений всех рёбер в графе на противоположные (двунаправленное ребро остаётся самим собой)

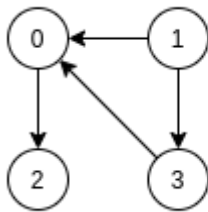
Такт DFS из вершины v - обход (в глубину) всех вершин графа, достижимых из v . Такт можно интерпретировать как рекурсивный вызов функции. Такт обработки вершины, у которой нет соседей, будет равняться 1.

Время выхода вершины -- число, соответствующее времени выхода рекурсии алгоритма DFS из вершины. Притом, изначально счётчик времени 0, увеличивается он лишь в двух случаях:

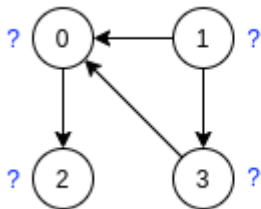
1. Начало нового такта DFS
2. Прохождение по ребру (при том, не важно, рекурсивный проход или нет)

Пример присвоения времени выхода:

Дан следующий граф:



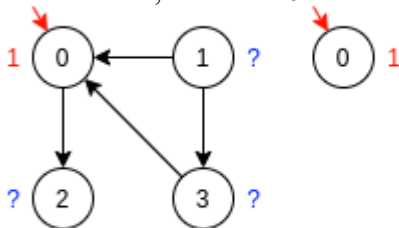
Обойдём его алгоритмом DFS, помечая вершины, согласно правилам выше:



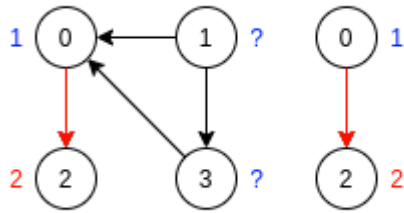
Непройденные вершины обозначим '?'

Будем выбирать стартовую вершину в порядке возрастания непомеченных вершин

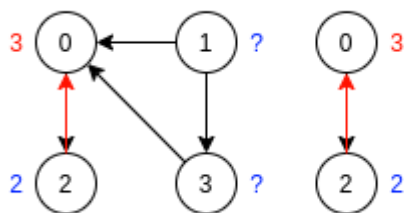
Изначально, счётчик 0



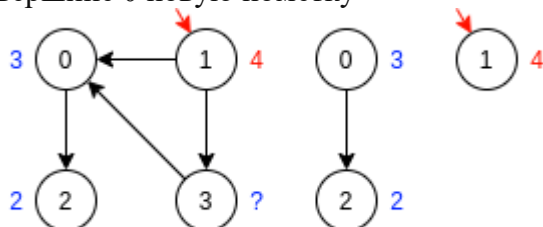
Начинаем новый такт, выбрав вершину 0 (как вершину с минимальным индексом).
 Повышаем счётчик на 1 и присваиваем это число метке вершины 0.
 Попутно будем строить деревья тактов DFS справа от исходного графа



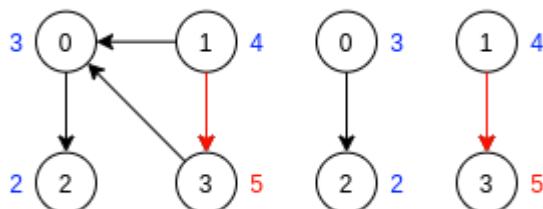
У вершины 0 есть лишь один сосед -- вершина 2. Проходим по ребру 0-2, повышая счётчик и присваивая метку вершине 2



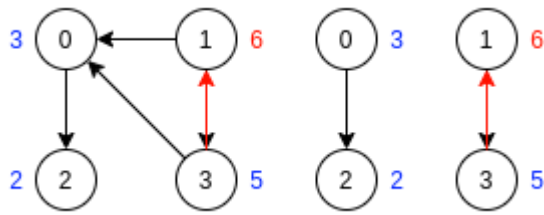
У вершины 2 нет соседей. Поэтому нам остаётся лишь рекурсивно вернуться в предыдущую вершину (чтобы понять, какая вершина была предыдущей, мы и строим дерево). Проходя по рекурсивному ребру 2-0, увеличиваем счётчик на 1 и присваиваем вершине 0 новую пометку



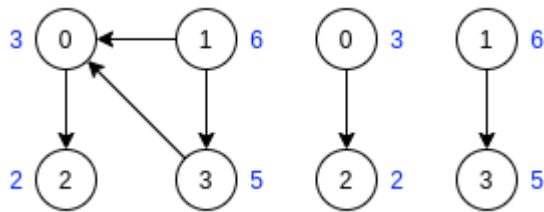
У вершины 0 больше нет непометченных соседей -- такт заканчивается. Начиная новый такт, увеличиваем счётчик и присваиваем следующей (в порядке возрастания) вершине метку счётчика. Также, начинаем строить дерево нового такта



У вершины 1 есть 2 соседа: 0 и 3. Но вершина 0 уже имеет метку -- в неё идти нельзя. Тогда идём в вершину 3, увеличивая счётчик и присваивая метку



У вершины 5 нет непомеченных соседей. Возвращаемся в предыдущую вершину, увеличивая счётчик, и присваивая новую метку



У вершины 1 больше нет соседей -- такт завершается. Также видим, что все вершины помечены. Алгоритм поиска в глубину завершён

Из примера прослеживается, что вершина начала такта (корень) имеет самую большую (в такте) метку - это объясняется рекурсивностью алгоритма DFS - алгоритм заходит в одну вершину и выходит из неё же. Этот факт понадобится при доказательстве.

Эйлеров путь/цикл

Определение. *Эйлеров путь* — это путь в графе, проходящий через все его рёбра.

Определение. *Эйлеров цикл* — это эйлеров путь, являющийся циклом.

Теорема . *Эйлеров цикл* в связном орграфе существует тогда и только тогда, когда у каждой его вершины число входящих в нее ребер равно числу выходящих.

Сначала проверим, существует ли эйлеров путь. Затем найдём все простые циклы и объединим их в один - это и будет эйлеровым циклом. Если граф таков, что эйлеров путь не является циклом, то, добавим недостающее ребро, найдём эйлеров цикл, потом удалим лишнее ребро.

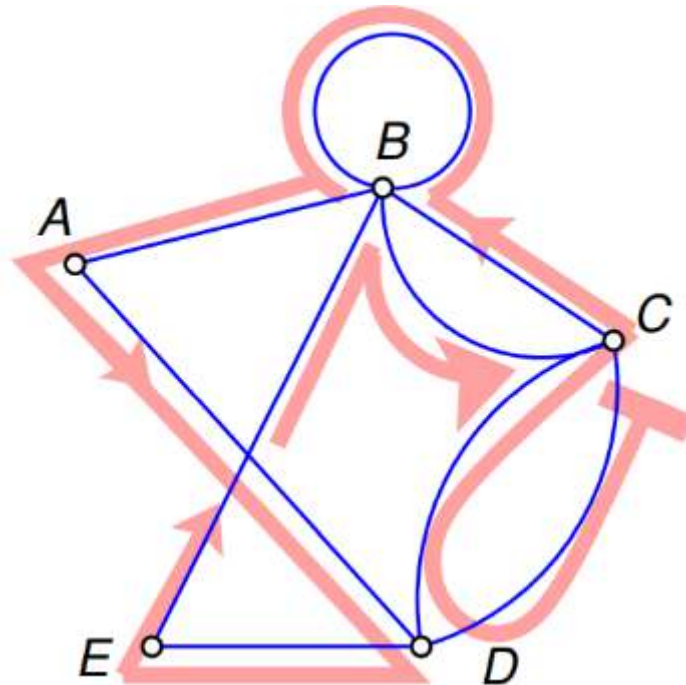
Чтобы проверить, существует ли эйлеров путь, нужно воспользоваться следующей теоремой. Эйлеров цикл существует тогда и только тогда, когда степени всех вершин чётны. Эйлеров путь существует тогда и только тогда, когда количество вершин с нечётными степенями равно двум (или нулю, в случае существования эйлерова цикла).

Кроме того, конечно, граф должен быть достаточно связным (т.е. если удалить из него все изолированные вершины, то должен получиться связный граф).

Искать все циклы и объединять их будем одной рекурсивной процедурой:

procedure FindEulerPath (V)

1. перебрать все рёбра, выходящие из вершины V;
каждое такое ребро удаляем из графа, и
вызываем FindEulerPath из второго конца этого ребра;
2. добавляем вершину V в ответ.



Граф на пяти вершинах и один из его эйлеровых циклов: CDCBBADEBC

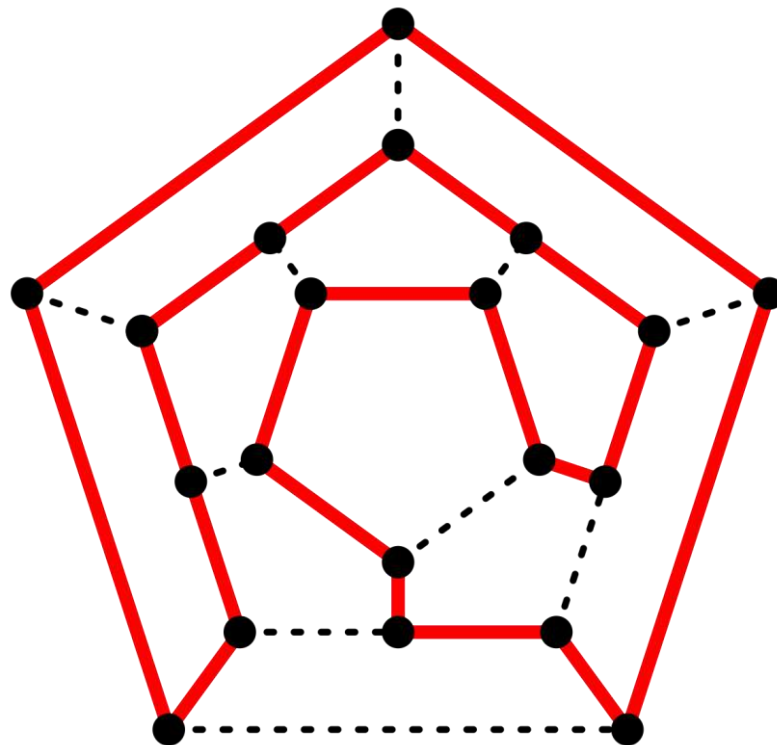
Гамильтонов цикл

Определение: *Гамильтоновым путём* (англ. *Hamiltonian path*) называется простой путь, проходящий через каждую вершину графа ровно один раз.

Определение: *Гамильтоновым циклом* (англ. *Hamiltonian cycle*) называют замкнутый гамильтонов путь.

Определение: Граф называется гамильтоновым (англ. *Hamiltonian graph*), если он содержит гамильтонов цикл.

Теорема Дирака: Пусть G — неориентированный граф и δ — минимальная степень его вершин. Если $n \geq 3$ и $\delta \geq n/2$, то G — гамильтонов граф.



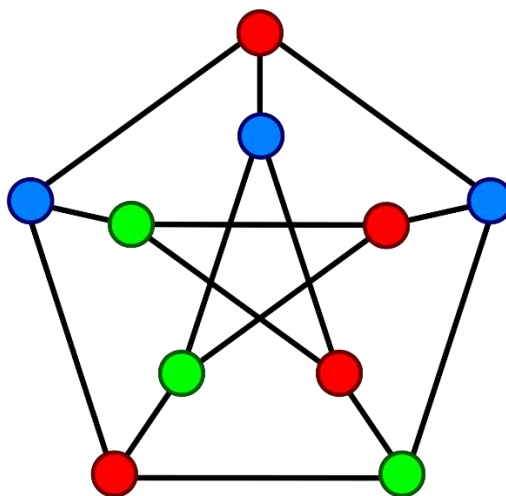
Пример Гамильтонова цикла

Раскраска графа

Раскраской вершин графа называется назначение цветов его вершинам. Обычно цвета - это числа $1, 2 \dots k$. Тогда раскраска является функцией, определенной на множестве вершин графа и принимающей значения в множестве $\{1, 2 \dots k\}$. Раскраску можно также рассматривать как разбиение множества вершин $V = V_1 \cup V_2 \cup \dots \cup V_k$, где V_i - множество вершин цвета i . Множества V_i называют цветными классами. Раскраска называется правильной, если каждый цветной класс является независимым множеством. Иначе говоря, в правильной раскраске любые две смежные вершины должны иметь разные цвета. Задача о раскраске состоит в нахождении правильной раскраски данного графа G в наименьшее число цветов. Это число называется хроматическим числом графа и обозначается $\chi(G)$.

В правильной раскраске полного графа K_n все вершины должны иметь разные цвета, поэтому $\chi(K_n) = n$. Если в каком-нибудь графе имеется полный подграф с k вершинами, то для раскраски этого подграфа необходимо k цветов. Отсюда следует, что для любого графа выполняется неравенство

$$\chi(G) \geq \omega(G).$$



Минимальное остовное дерево у неорграфа

Дан взвешенный неориентированный граф G с n вершинами и m рёбрами. Требуется найти такое поддерево этого графа, которое бы соединяло все его вершины, и при этом обладало наименьшим возможным весом (т.е. суммой весов рёбер). Поддерево — это набор рёбер, соединяющих все вершины, причём из любой вершины можно добраться до любой другой ровно одним простым путём.

Такое поддерево называется минимальным остовным деревом или просто **минимальным остовом**. Легко понять, что любой остов обязательно будет содержать $n - 1$ ребро.

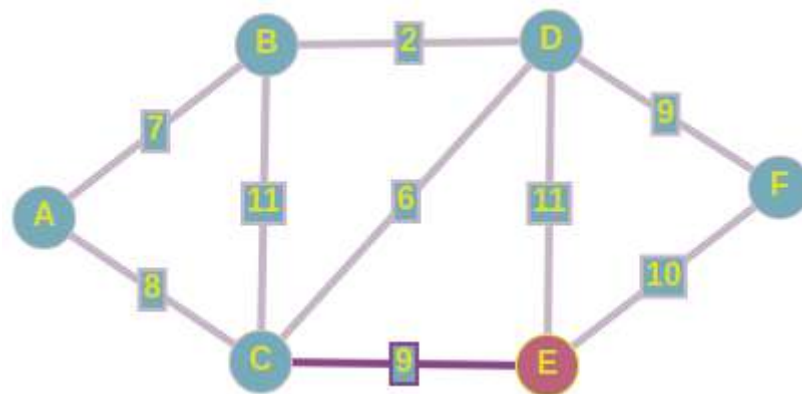
Алгоритм Прима

Суть самого алгоритма Прима тоже сводится к жадному перебору рёбер, но уже из определенного множества. На входе так же имеется пустой подграф, который и будем достраивать до потенциального минимального остовного дерева.

- Изначально наш подграф состоит из одной любой вершины исходного графа.
- Затем из рёбер инцидентных этой вершине, выбирается такое минимальное ребро, которое связала бы две абсолютно разные компоненты связности, одной из которых и является наш подграф. То есть, как только у нас появляется возможность добавить новую вершину в наш подграф, мы тут же включаем ее по минимальному возможному весу.
- Продолжаем выполнять предыдущий шаг до тех пор, пока не найдем искомое MST.

Разбор конкретного примера

Выбираем чисто случайно вершину E, далее рассмотрим все ребра исходящие из нее, включаем в наше остовное дерево ребро $C \leftrightarrow E$; $w = 9$, так как данное ребро имеет минимальный вес из всех рёбер инцидентных множеству вершин нашего подграфа. Имеем следующее:



Подграф после добавления 1-го ребра

Теперь выборка производится из рёбер:

$D \leftrightarrow C$; $w = 6$

$A \leftrightarrow C$; $w = 8$

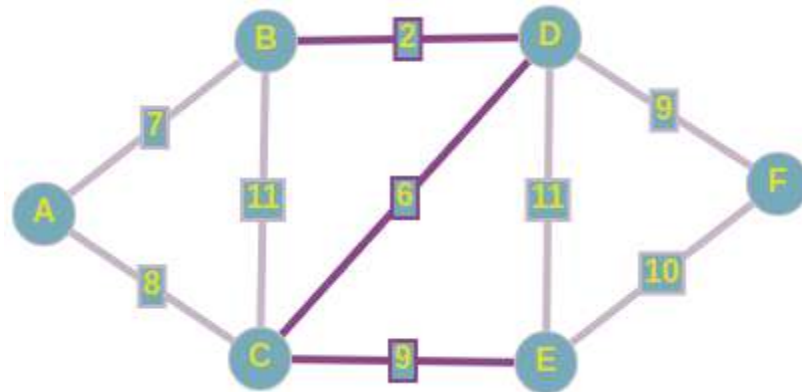
$F \leftrightarrow E$; $w = 10$

$B \leftrightarrow C$; $w = 11$

$D \leftrightarrow E$; $w = 11$

То есть, в данный момент, мы знаем только о двух вершинах, соответственно, знаем о всех ребрах, исходящих из них. Про связи между другими вершинами, которые не включены в наш подграф, мы ничего не знаем, поэтому они на этом шаге не рассматриваются.

Добавляем в наш подграф ребро $D \leftrightarrow C$ и по аналогии добавляем ребро $D \leftrightarrow B$. Получаем следующее:

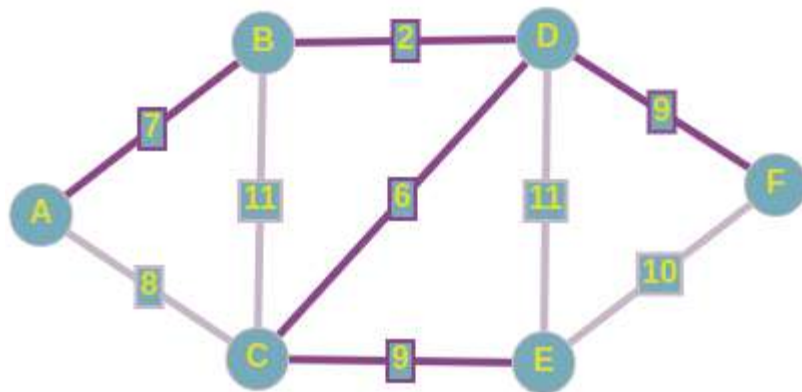


Подграф, полученный после добавления рассмотренных рёбер

Давайте добьем наш подграф до минимального остовного дерева. Вы, наверное, уже догадались о том, по каким ребрам мы будем связывать наши оставшиеся вершины:

А и F.

Проводим последние штрихи и получили тот же самый подграф в качестве минимального остовного дерева. Но как мы ранее говорили, сам подграф ничего не решает, главное тут - множество рёбер, которые включены в наше остовное дерево.

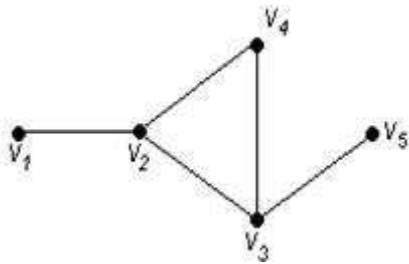


Суммарный вес искомого MST равен 33.

Центр, радиус, диаметр и медиана

Определение. Диаметр связного графа называется максимально возможное расстояние между двумя его вершинами.

Определение. Центром графа называется такая вершина, что максимальное расстояние между ней и любой другой вершиной является наименьшим из всех возможных; это расстояние называется **радиусом** графа.



$R(G) = 2$ и $D(G) = 3$, центрами являются вершины v_2, v_3, v_4 .

Определение: Медиана — вершина графа, у которой сумма кратчайших расстояний от неё до вершин графа минимальная.

Алгоритм Дейкстры

Алгоритм находит кратчайшие пути от одной из вершин графа до всех остальных. Алгоритм работает только для графов без рёбер отрицательного веса.

Каждой вершине из V сопоставим метку — минимальное известное расстояние от этой вершины до a .

Алгоритм работает пошагово — на каждом шаге он «посещает» одну вершину и пытается уменьшать метки.

Работа алгоритма завершается, когда все вершины посещены.

Инициализация.

Метка самой вершины a полагается равной 0, метки остальных вершин — бесконечности.

Это отражает то, что расстояния от a до других вершин пока неизвестны.

Все вершины графа помечаются как непосещённые.

Шаг алгоритма.

Если все вершины посещены, алгоритм завершается.

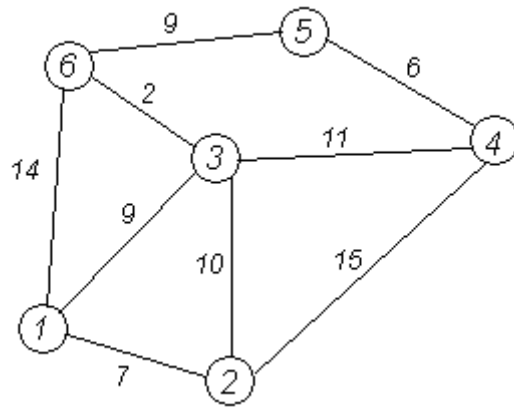
В противном случае, из ещё не посещённых вершин выбирается вершина u , имеющая минимальную метку.

Мы рассматриваем всевозможные маршруты, в которых u является предпоследним пунктом. Вершины, в которые ведут рёбра из u , назовём *соседями* этой вершины. Для каждого соседа вершины u , кроме отмеченных как посещённые, рассмотрим новую длину пути, равную сумме значений текущей метки u и длины ребра, соединяющего u с этим соседом.

Если полученное значение длины меньше значения метки соседа, заменим значение метки полученным значением длины. Рассмотрев всех соседей, пометим вершину u как посещённую и повторим [шаг алгоритма](#).

Пример: Рассмотрим выполнение алгоритма на примере графа, показанного на рисунке.

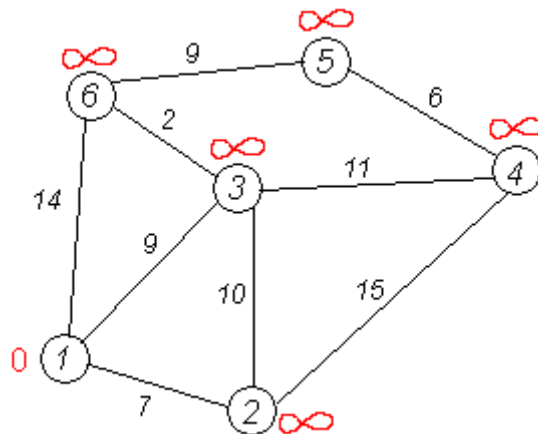
Пусть требуется найти кратчайшие расстояния от 1-й вершины до всех остальных.



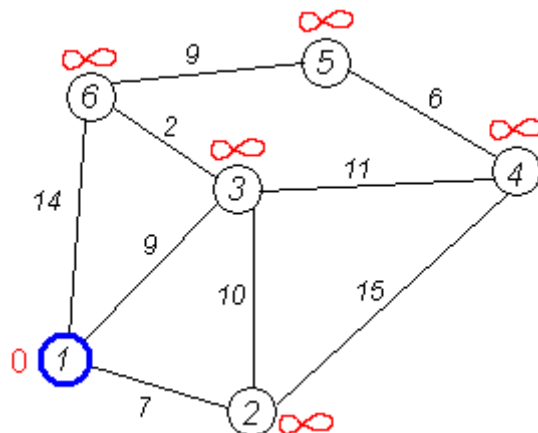
Кружками обозначены вершины, линиями — пути между ними (рёбра графа).

В кружках обозначены номера вершин, над рёбрами обозначен их вес — длина пути.

Рядом с каждой вершиной красным обозначена метка — длина кратчайшего пути в эту вершину из вершины 1.



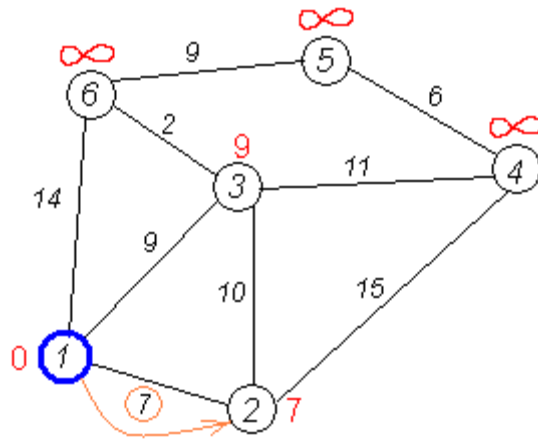
Минимальную метку имеет вершина 1. Её соседями являются вершины 2, 3 и 6.



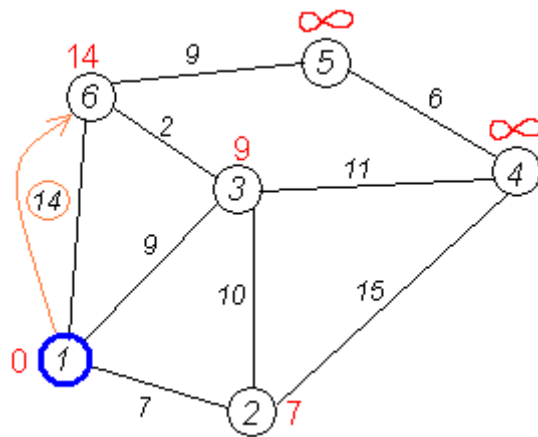
Первый по очереди сосед вершины 1 — вершина 2, потому что длина пути до неё минимальна.

Длина пути в неё через вершину 1 равна сумме значения метки вершины 1 и длины ребра, идущего из 1-й в 2-ю, то есть $0 + 7 = 7$.

Это меньше текущей метки вершины 2, бесконечности, поэтому новая метка 2-й вершины равна 7.



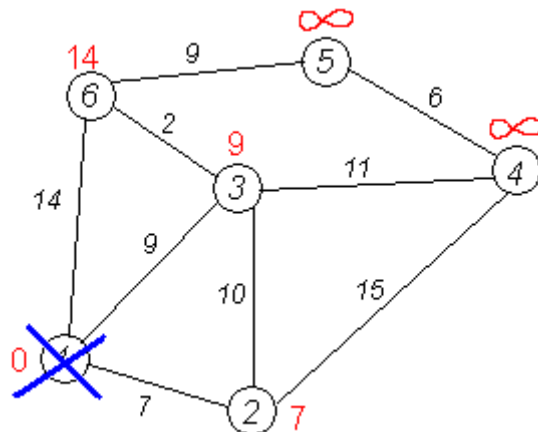
Аналогичную операцию проделываем с двумя другими соседями 1-й вершины — 3-й и 6-й.



Все соседи вершины 1 проверены.

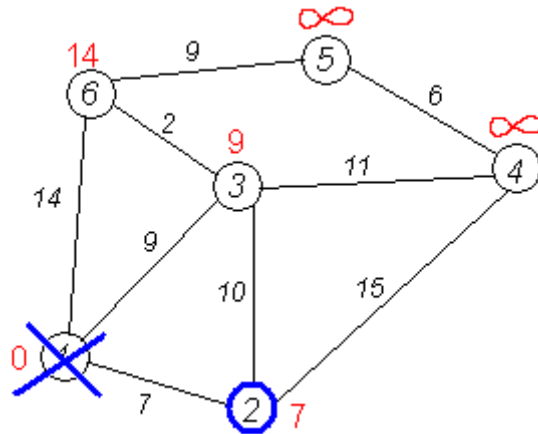
Текущее минимальное расстояние до вершины 1 считается окончательным и пересмотру не подлежит.

Вычеркнем её из графа, чтобы отметить, что эта вершина посещена.



Второй шаг.

Снова находим «ближайшую» из непосещённых вершин. Это вершина 2 с меткой 7.

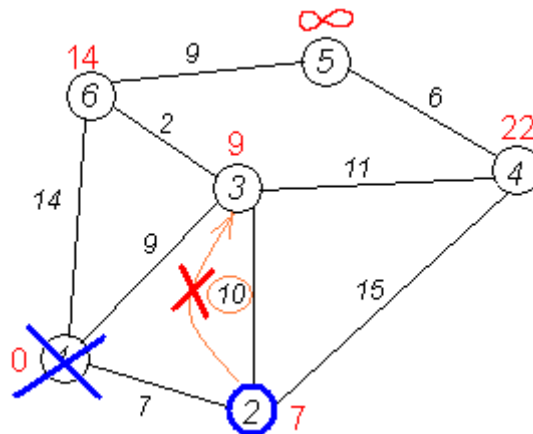


Снова пытаемся уменьшить метки соседей выбранной вершины, пытаясь пройти в них через 2-ю вершину. Соседями вершины 2 являются вершины 1, 3 и 4.

Первый (по порядку) сосед вершины 2 — вершина 1. Но она уже посещена, поэтому с 1-й вершиной ничего не делаем.

Следующий сосед — вершина 3, так как имеет минимальную метку.

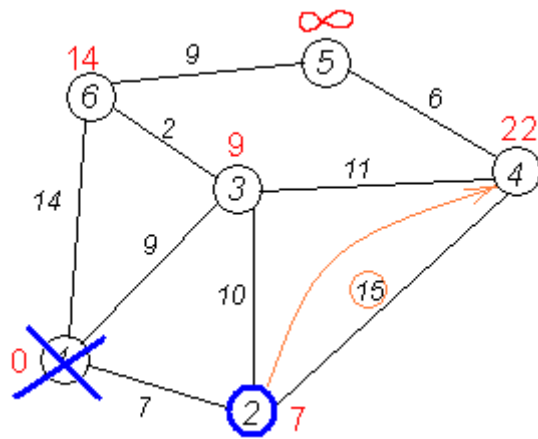
Если идти в неё через 2, то длина такого пути будет равна 17 ($7 + 10 = 17$). Но текущая метка третьей вершины равна 9, а это меньше 17, поэтому метка не меняется.



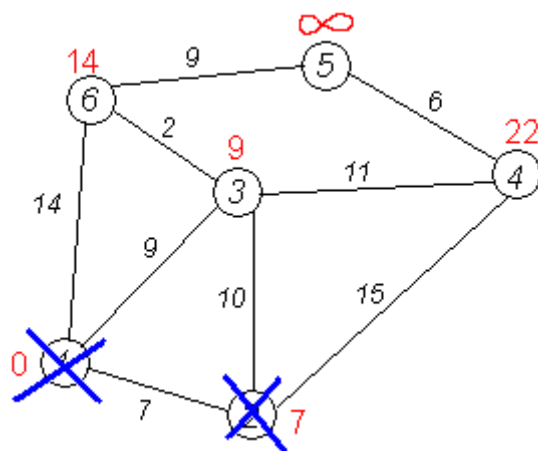
Ещё один сосед вершины 2 — вершина 4.

Если идти в неё через 2-ю, то длина такого пути будет равна сумме кратчайшего расстояния до 2-й вершины и расстояния между вершинами 2 и 4, то есть 22 ($7 + 15 = 22$).

Поскольку $22 < \infty$, устанавливаем метку вершины 4 равной 22.

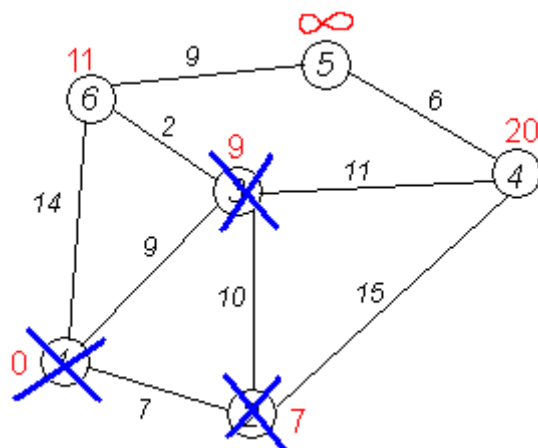


Все соседи вершины 2 просмотрены, замораживаем расстояние до неё и помечаем её как посещённую.



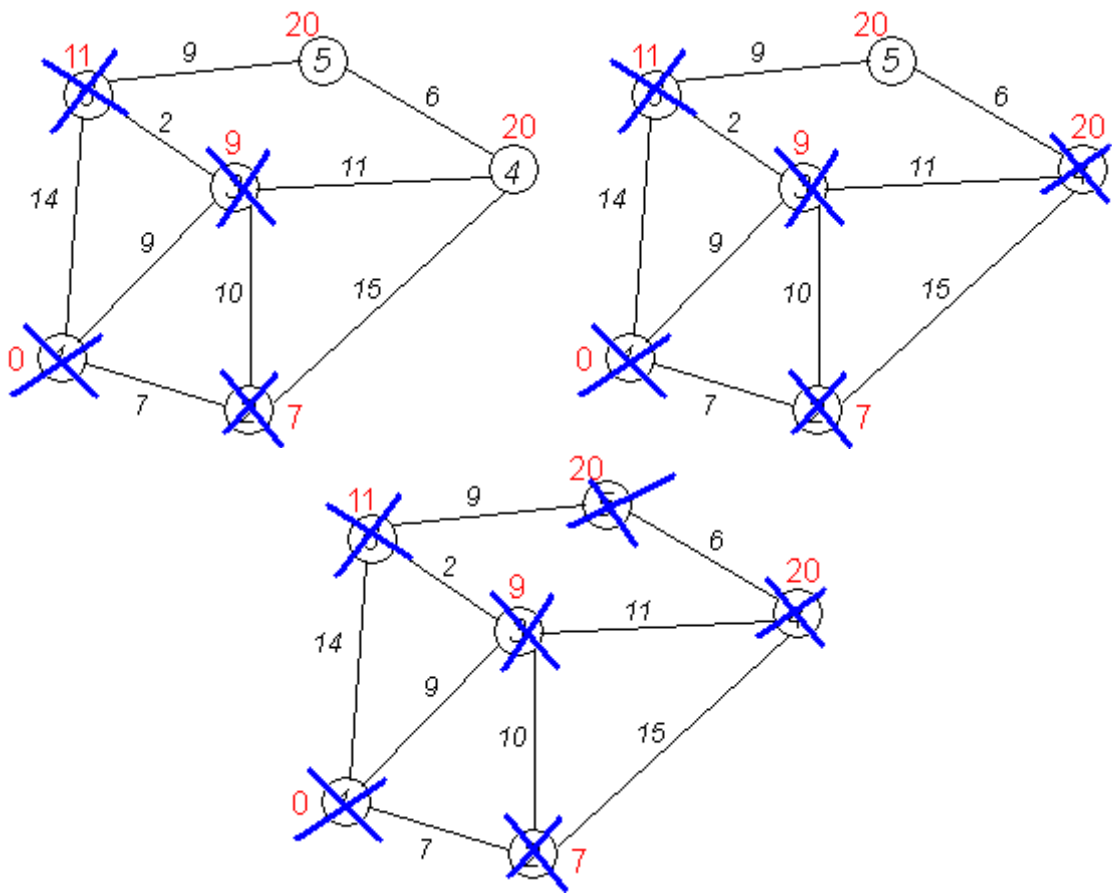
Третий шаг.

Повторяем шаг алгоритма, выбрав вершину 3. После её «обработки» получим такие результаты:



Дальнейшие шаги.

Повторяем шаг алгоритма для оставшихся вершин. Это будут вершины 6, 4 и 5, соответственно порядку.



Завершение выполнения алгоритма.

Алгоритм заканчивает работу, когда все вершины посещены.

Результат работы алгоритма виден на последнем рисунке: кратчайший путь от вершины 1 до 2-й составляет 7, до 3-й — 9, до 4-й — 20, до 5-й — 20, до 6-й — 11.

Если в какой-то момент все непосещённые вершины помечены бесконечностью, то это значит, что до этих вершин нельзя добраться (то есть граф несвязный). Тогда алгоритм может быть завершён досрочно.

Алгоритм Флойда

Алгоритм Флойда — Уоршелла — алгоритм для нахождения кратчайших расстояний между всеми вершинами взвешенного графа без циклов с отрицательными весами.

Описание алгоритма

Ключевая идея алгоритма — разбиение процесса поиска кратчайших путей на **фазы**.

Перед k -ой фазой ($k = 1 \dots n$) считается, что в матрице расстояний $d[i][j]$ сохранены длины таких кратчайших путей, которые содержат в качестве внутренних вершин только вершины из множества $\{1, 2, \dots, k-1\}$ (вершины графа мы нумеруем, начиная с единицы).

Иными словами, перед k -ой фазой величина $d[i][j]$ равна длине кратчайшего пути из вершины i в вершину j , если этому пути разрешается заходить только в вершины с номерами, меньшими k (начало и конец пути не считаются).

Легко убедиться, что чтобы это свойство выполнилось для первой фазы, достаточно в матрицу расстояний $d[i][j]$ записать матрицу смежности графа: $d[i][j] = g[i][j]$ — стоимости ребра из вершины i в вершину j . При этом, если между какими-то вершинами ребра нет, то записать следует величину "бесконечность" ∞ . Из вершины в саму себя всегда следует записывать величину 0, это критично для алгоритма.

Пусть теперь мы находимся на k -ой фазе, и хотим **пересчитать** матрицу $d[i][j]$ таким образом, чтобы она соответствовала требованиям уже для $k+1$ -ой фазы. Зафиксируем какие-то вершины i и j . У нас возникает два принципиально разных случая:

- Кратчайший путь из вершины i в вершину j , которому разрешено дополнительно проходить через вершины $\{1, 2, \dots, k\}$, **совпадает** с кратчайшим путём, которому разрешено проходить через вершины множества $\{1, 2, \dots, k-1\}$.

В этом случае величина $d[i][j]$ не изменится при переходе с k -ой на $k+1$ -ую фазу.

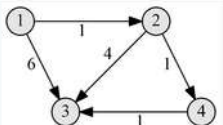
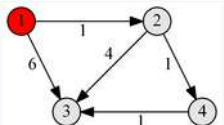
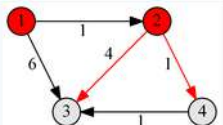
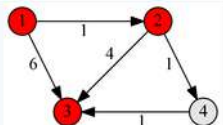
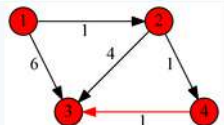
- "Новый" кратчайший путь стал **лучше** "старого" пути.

Это означает, что "новый" кратчайший путь проходит через вершину k . Сразу отметим, что мы не потеряем общности, рассматривая далее

только простые пути (т.е. пути, не проходящие по какой-то вершине дважды).

Тогда заметим, что если мы разобьём этот "новый" путь вершиной k на две половинки (одна идущая $i \Rightarrow k$, а другая — $k \Rightarrow j$), то каждая из этих половинок уже не заходит в вершину k . Но тогда получается, что длина каждой из этих половинок была посчитана ещё на $k - 1$ -ой фазе или ещё раньше, и нам достаточно взять просто сумму $d[i][k] + d[k][j]$, она и даст длину "нового" кратчайшего пути.

Пример работы:

| $i = 0$ | $i = 1$ | $i = 2$ | $i = 3$ | $i = 4$ |
|---|---|--|--|--|
|  |  |  |  |  |
| $\begin{pmatrix} \times & 1 & 6 & \infty \\ \infty & \times & 4 & 1 \\ \infty & \infty & \times & \infty \\ \infty & \infty & 1 & \times \end{pmatrix}$ | $\begin{pmatrix} \times & 1 & 6 & \infty \\ \infty & \times & 4 & 1 \\ \infty & \infty & \times & \infty \\ \infty & \infty & 1 & \times \end{pmatrix}$ | $\begin{pmatrix} \times & 1 & 5 & 2 \\ \infty & \times & 4 & 1 \\ \infty & \infty & \times & \infty \\ \infty & \infty & 1 & \times \end{pmatrix}$ | $\begin{pmatrix} \times & 1 & 5 & 2 \\ \infty & \times & 4 & 1 \\ \infty & \infty & \times & \infty \\ \infty & \infty & 1 & \times \end{pmatrix}$ | $\begin{pmatrix} \times & 1 & 3 & 2 \\ \infty & \times & 2 & 1 \\ \infty & \infty & \times & \infty \\ \infty & \infty & 1 & \times \end{pmatrix}$ |