Задача о кратчайшем пути

Задача о кратчайшем пути — задача поиска самого короткого пути (цепи) между двумя вершинами в графе, в которой сумма весов рёбер (дуг), минимальна.

Задача о кратчайшем пути является одной из важнейших классических задач теории графов. Сегодня известно множество алгоритмов для её решения.

- Алгоритм Дейкстры находит кратчайший путь от одной из вершин графа до всех остальных. Алгоритм работает только для графов без рёбер отрицательного веса.
- Алгоритм Беллмана — Форда находит кратчайшие пути от одной вершины графа до всех остальных во взвешенном графе. Вес рёбер может быть отрицательным.
- Алгоритм Флойда — Уоршелла находит кратчайшие пути между всеми вершинами взвешенного ориентированного графа.
- Алгоритм Джонсона находит кратчайшие пути между всеми парами вершин взвешенного ориентированного графа.
- Алгоритм поиска A* находит маршрут с наименьшей стоимостью от одной вершины (начальной) к другой (целевой, конечной), используя алгоритм поиска по первому наилучшему совпадению на графе.
- Алгоритм Ли (волновой алгоритм) основан на методе поиска в ширину. Находит путь между вершинами s и t графа (s не совпадает с t), содержащий минимальное количество промежуточных вершин (рёбер).

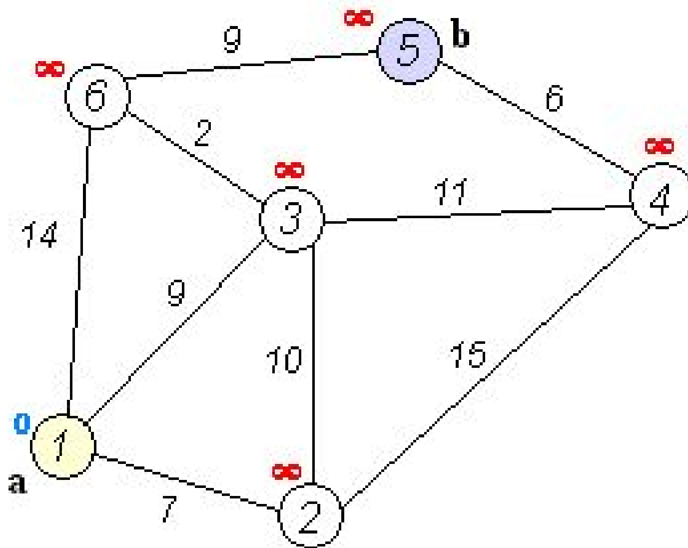
Алгоритм Дейкстры

Алгори́тм Де́йкстры (англ. *Dijkstra's algorithm*) — алгоритм на графах, изобретённый нидерландским учёным Эдсгером Дейкстрой в 1959 году. Находит кратчайшие пути от одной из вершин графа до всех остальных. Алгоритм работает только для графов без рёбер отрицательного веса. Алгоритм широко применяется в программировании и технологиях, например, его используют протоколы маршрутизации OSPF и IS-IS.

Сложность, зависит от реализации и в худшем случае составляет O(n^2).
Если использовать бинарную кучу, то O(n log n + m log n)
Если использовать фибначчиеву кучу, то O(n log n + log n)

“скрытые константы в асимптотических оценках трудоёмкости велики и использование фибоначчиевых куч редко оказывается целесообразным: обычные двоичные кучи на практике эффективнее”

Идея алгоритма



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.

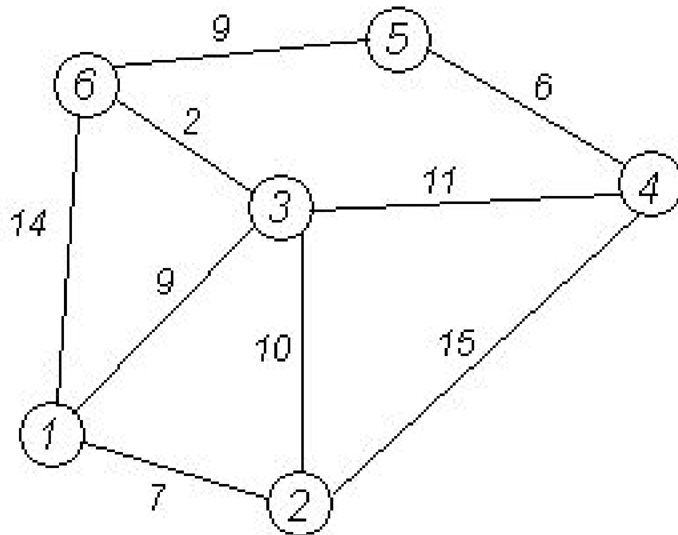
Инициализация. Метка самой вершины ***a*** полагается равной 0, метки остальных вершин — бесконечности. Это отражает то, что расстояния от ***a*** до других вершин пока неизвестны. Все вершины графа помечаются как непосещённые.

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.

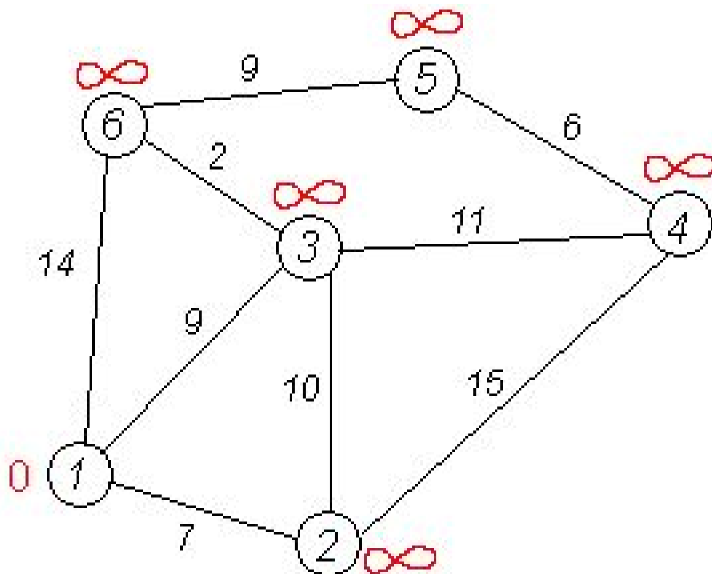
Пример

Рассмотрим выполнение алгоритма на примере графа, показанного на рисунке.

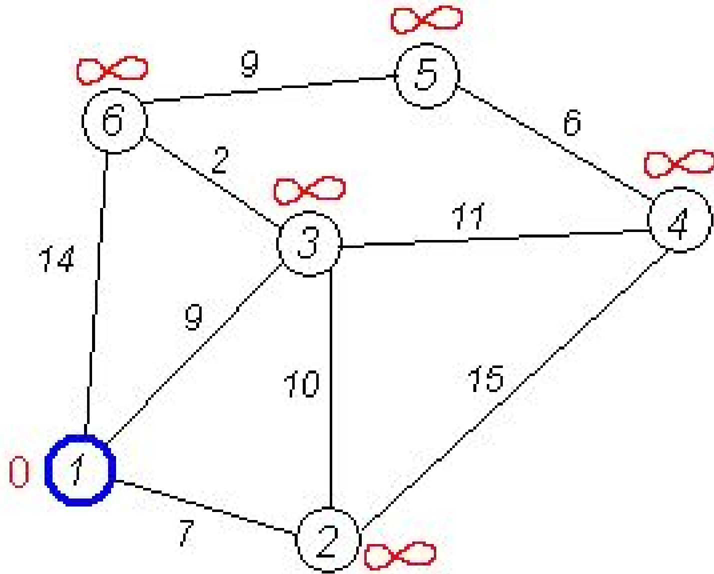
Пусть требуется найти кратчайшие расстояния от 1-й вершины до всех остальных.



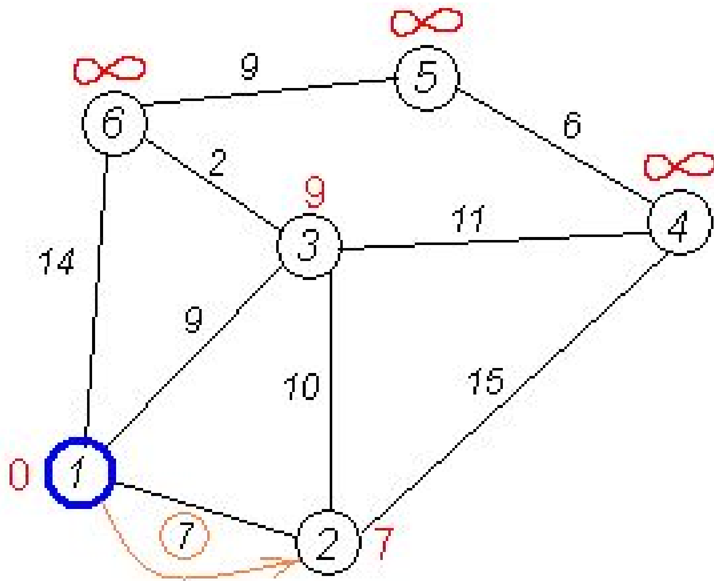
Кружками обозначены вершины, линиями — пути между ними (рёбра графа). В кружках обозначены номера вершин, над рёбрами обозначен их вес — длина пути. Рядом с каждой вершиной красным обозначена метка — длина кратчайшего пути в эту вершину из вершины 1.



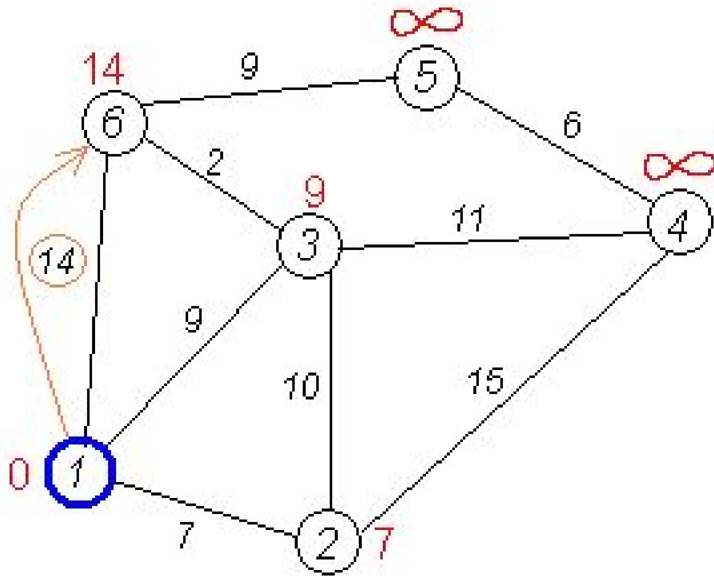
Первый шаг. Рассмотрим шаг алгоритма Дейкстры для нашего примера. Минимальную метку имеет вершина 1. Её соседями являются вершины 2, 3 и 6.



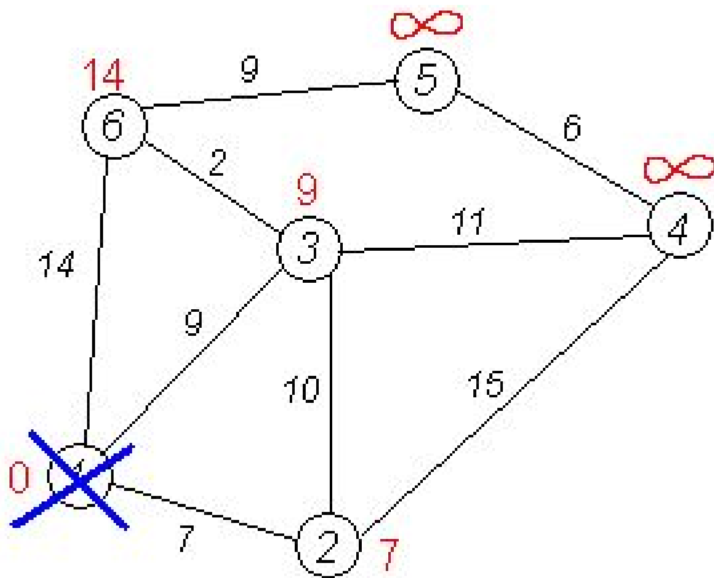
Первый по очереди сосед вершины 1 — вершина 2, потому что длина пути до неё минимальна. Длина пути в неё через вершину 1 равна сумме значения метки вершины 1 и длины ребра, идущего из 1-й в 2-ю, то есть 0 + 7 = 7. Это меньше текущей метки вершины 2, бесконечности, поэтому новая метка 2-й вершины равна 7.



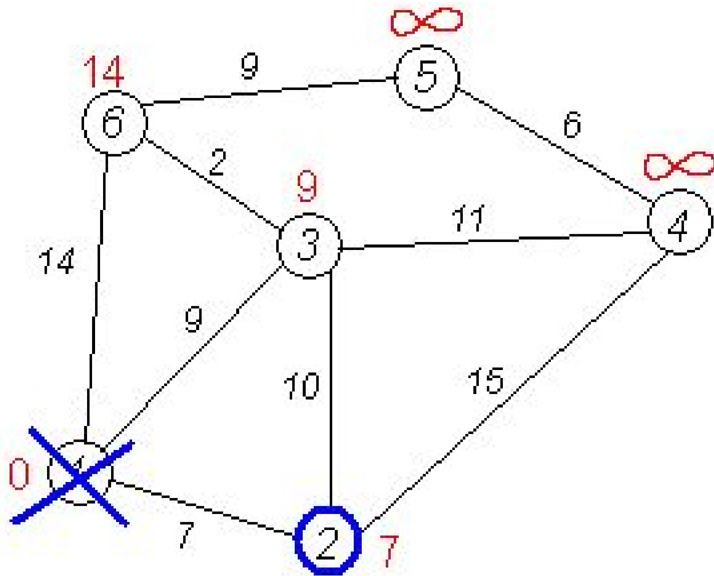
Аналогичную операцию проделываем с двумя другими соседями 1-й вершины — 3-й и 6-й.



Все соседи вершины 1 проверены. Текущее минимальное расстояние до вершины 1 считается окончательным и пересмотру не подлежит (то, что это действительно так, впервые доказал Э. Дейкстра). Вычеркнем её из графа, чтобы отметить, что эта вершина посещена.



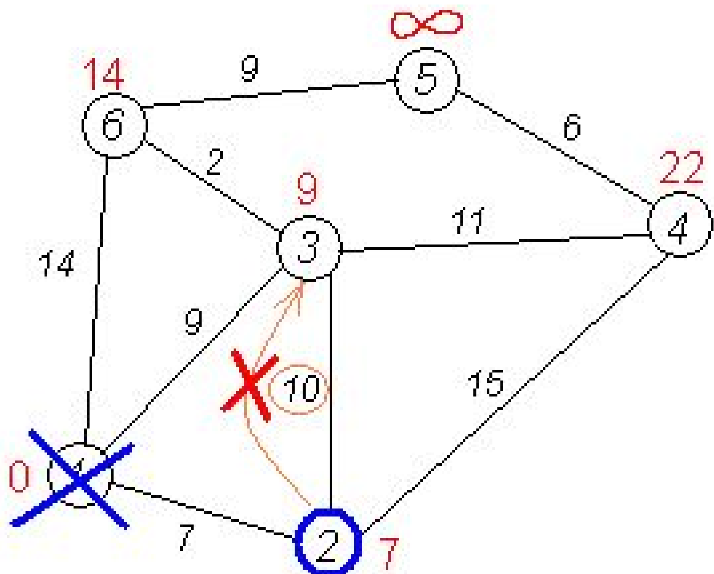
Второй шаг. Шаг алгоритма повторяется. Снова находим «ближайшую» из непосещённых вершин. Это вершина 2 с меткой 7.



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

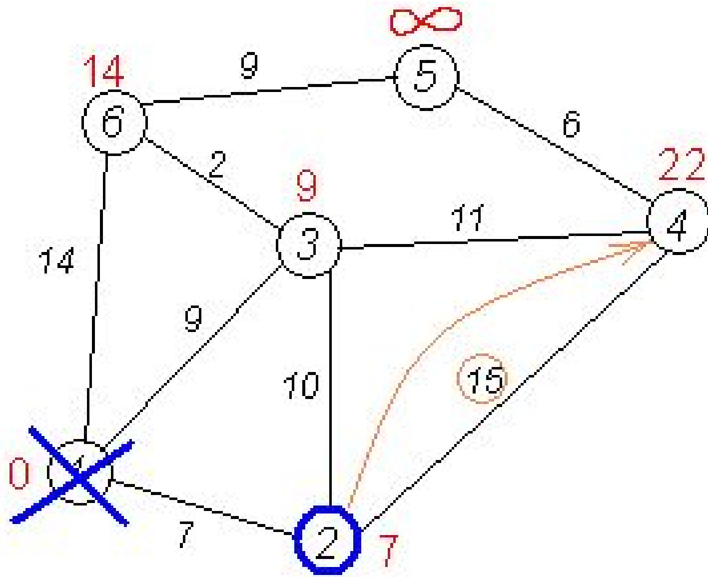
Первый (по порядку) сосед вершины 2 — вершина 1. Но она уже посещена, поэтому с 1-й вершиной ничего не делаем.

Следующий сосед вершины 2 — вершина 3, так как имеет минимальную метку из вершин, отмеченных как не посещённые. 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.

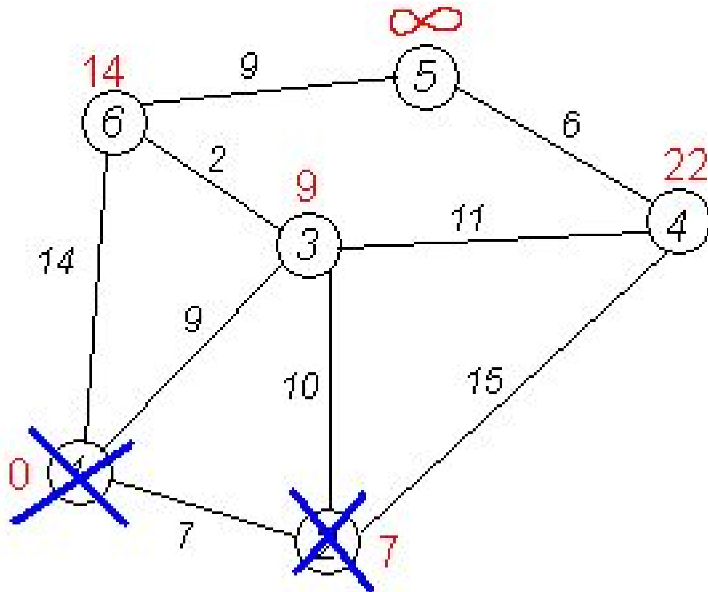


Ещё один сосед вершины 2 — вершина 4. Если идти в неё через 2-ю, то длина такого пути будет равна сумме кратчайшего расстояния до 2-й вершины и расстояния между вершинами

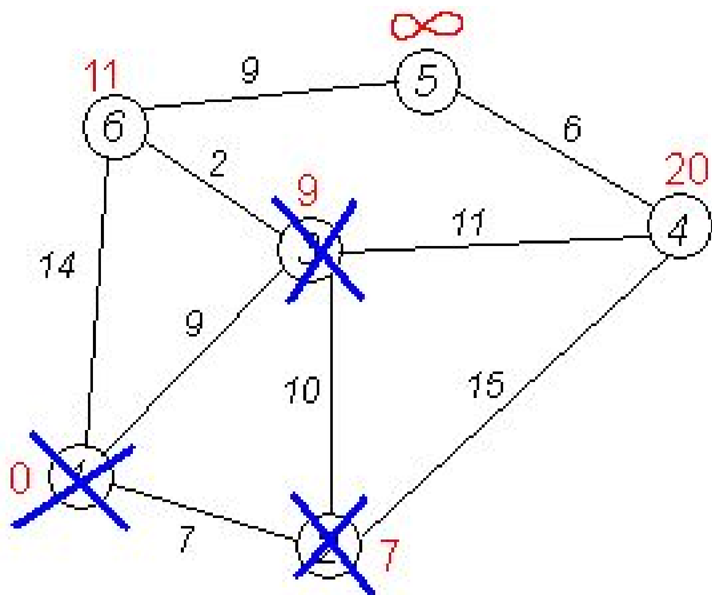
2 и 4, то есть 22 (7 + 15 = 22). Поскольку 22<{\displaystyle \infty } , устанавливаем метку вершины 4 равной 22.



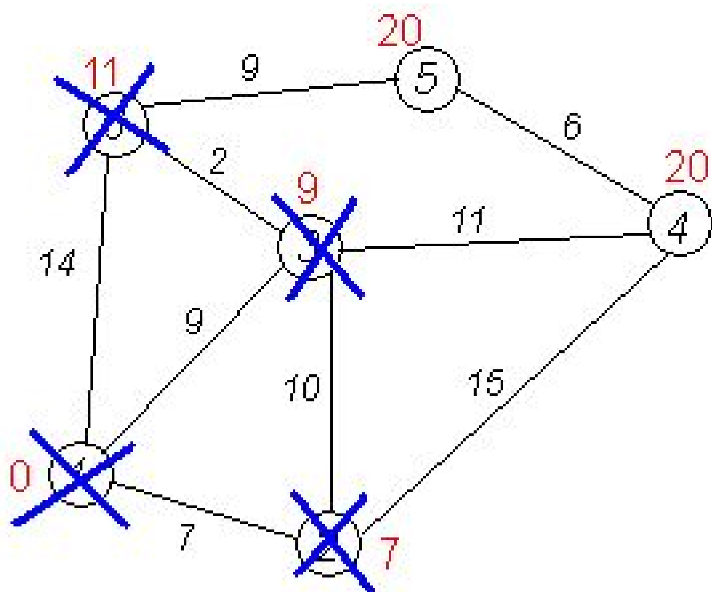
Все соседи вершины 2 просмотрены, замораживаем расстояние до неё и помечаем её как посещённую.

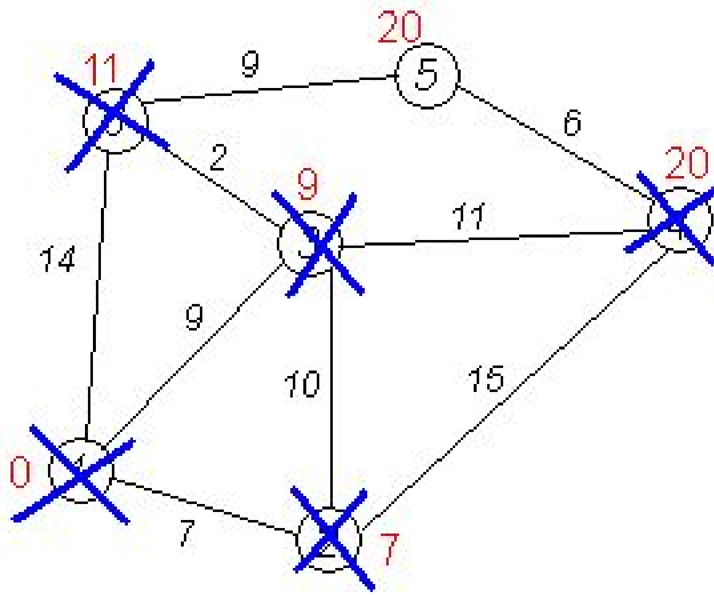


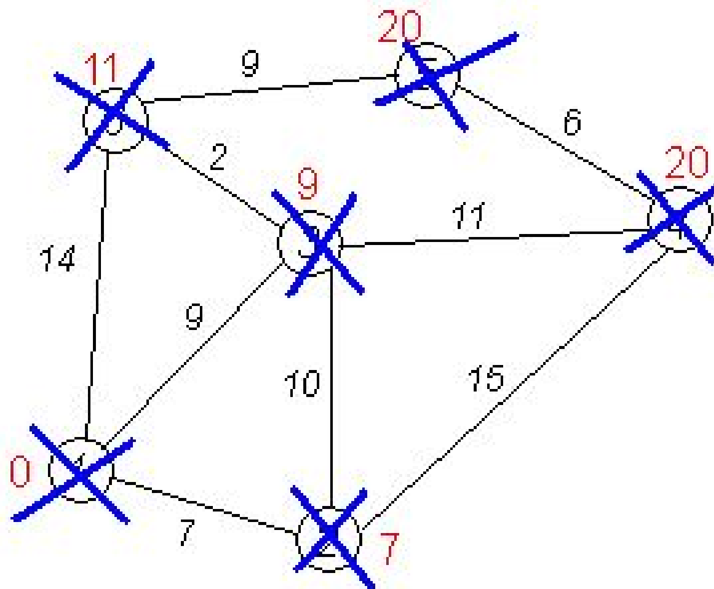
Третий шаг. Повторяем шаг алгоритма, выбрав вершину 3. После её «обработки» получим такие результаты:



Дальнейшие шаги. Повторяем шаг алгоритма для оставшихся вершин 6, 4 и 5.







Завершение выполнения алгоритма. Алгоритм заканчивает работу, когда нельзя больше обработать ни одной вершины. В данном примере все вершины зачёркнуты, однако ошибочно полагать, что так будет в любом примере — некоторые вершины могут остаться незачёркнутыми, если до них нельзя добраться, то есть если граф несвязный. 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.

Псевдокод

```
Дейкстра(A):
	для v из V
		метка[v] = макс_значение
	метка[A] = 0

	пока метка не пусто
		v = метка.МинимальноеЗначение()
		//----
		для e из v
			если метка.Содержит(e.B) то
				если метка[v]+e.вес < метка[e.B] то
					метка[e.B] = метка[v]+e.вес
					путь[e.B].добавить(e.B)
		дистанция[v] = метка[v]
		метка.Удалить(v)
```

```
class Edge {
	int A;
	int B;
	double вес;
}
```

Что такое мультиграф?

```
Путь(B):
	текущая = путь[B]
	пока текущая != A
		стек.Добавить(текущая)
		текущая = путь[текущая]
```

Как остановить Дейскру, если нужна дистанция только до B?

Алгоритм Беллмана — Форда

Алгоритм Беллмана — Форда — алгоритм поиска кратчайшего пути во взвешенном графе. За время O(|V| |E|) алгоритм находит кратчайшие пути от одной вершины графа до всех остальных. В отличие от алгоритма Дейкстры, алгоритм Беллмана — Форда

допускает рёбра с отрицательным весом. Предложен независимо Ричардом Беллманом и Лестером Фордом.

Алгоритм маршрутизации RIP (алгоритм Беллмана — Форда) был впервые разработан в 1969 году, как основной для сети ARPANET.

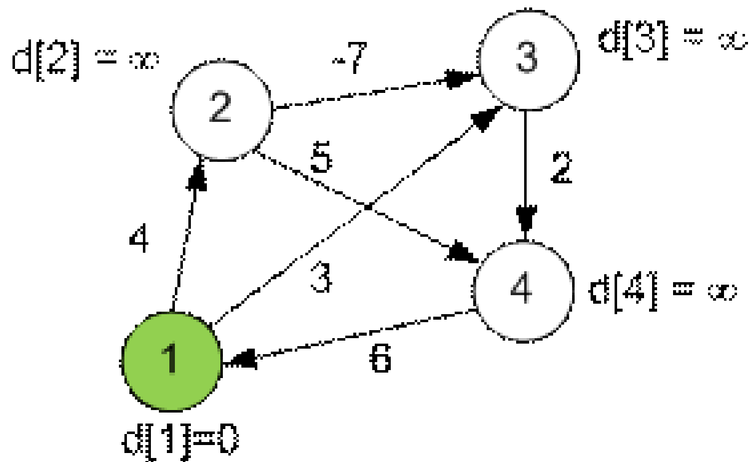
Сложность алгоритма Беллмана – Форда составляет O(VE).

Отрицательные циклы - это циклы, сумма весов рёбер которых отрицательна.

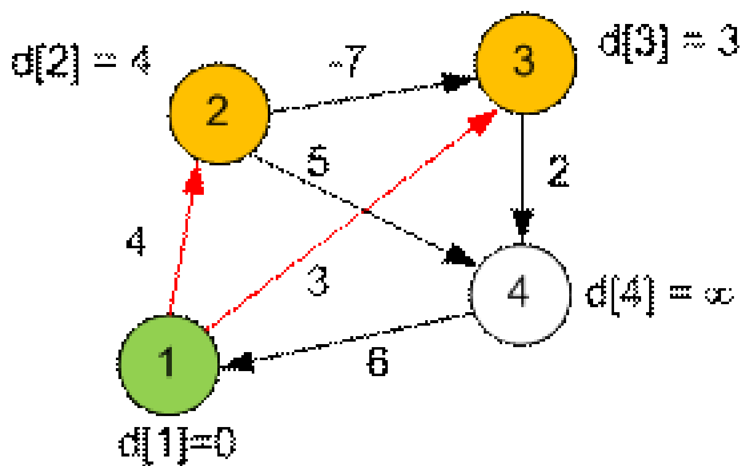
Алгоритм Беллмана-Форда представляет из себя несколько фаз. На каждой фазе просматриваются все рёбра графа, и алгоритм пытается произвести **релаксацию** (relax, ослабление) вдоль каждого ребра (a,b) стоимости c. Релаксация вдоль ребра — это попытка улучшить значение $d[b]$ значением $d[a]+c$. Фактически это значит, что мы пытаемся улучшить ответ для вершины b, пользуясь ребром (a,b) и текущим ответом для вершины a.

Утверждается, что достаточно $n-1$ фазы алгоритма, чтобы корректно рассчитать длины всех кратчайших путей в графе . Для недостижимых вершин расстояние $d[]$ останется равным бесконечности ∞.

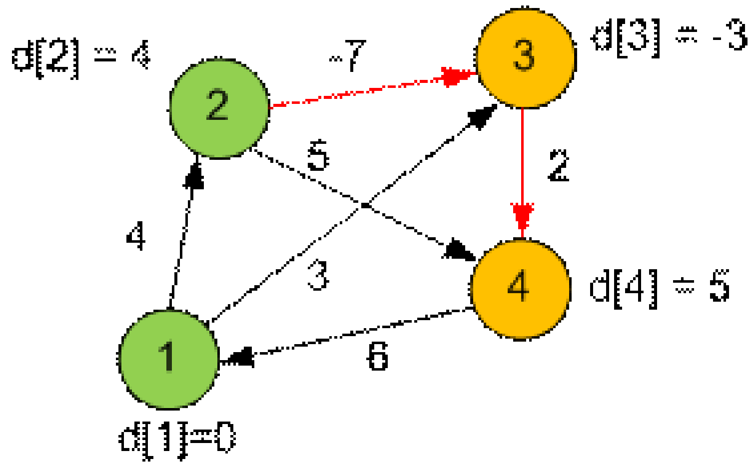
Пример. Промоделируем фазы алгоритма на приведенном ниже графе. Поскольку граф содержит 4 вершины, то достаточно выполнить 3 фазы алгоритма (в графе не существует вершин, находящихся друг от друга на расстоянии больше трех ребер).



На первой фазе алгоритма релаксируют ребра, исходящие из вершины 1:

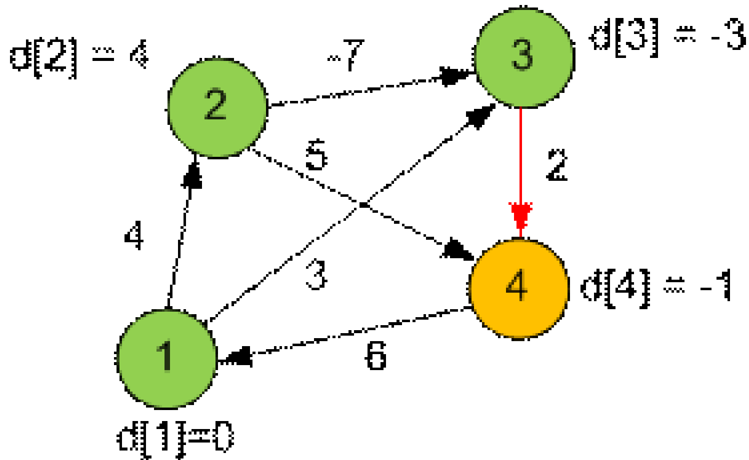


Пусть на второй фазе алгоритма релаксируют ребра (3, 4) и (2, 3) (именно в такой последовательности). Тогда сначала установится d[4] = d[3] + 2 = 3 + 2 = 5, а после релаксации ребра (2, 3) станет d[3] = d[2] – 7 = 4 – 7 = -3. Ребро (2, 4) релаксировать не будет, хотя если бы оно рассматривалось первым на второй фазе, то релаксация бы имела место.



На последней третьей фазе прорелаксирует только ребро (3, 4).

Значения d[*i*] содержат кратчайшие расстояние от начальной (первой) вершины до *i*-ой.




```
БеллманФорд(А):
    для v из V
        дистанция[v] = макс_значение
    дистанция[A] = 0

    для i=0 до n-1
```

```
для e=0 до m
  если дистанция[ребро[e].A] < макс_значение то
    если дистанция[ребро[e].B] <
         дистанция[ребро[e].A]+ребро[e].вес то
            дистанция[ребро[e].B] =
               дистанция[ребро[e].A]+ребро[e].вес
```

Зачем проверка если `дистанция[ребро[e].A < макс_значение` **то ?**
Какой аспект алгоритма позволяет работать с отрицательными весам?

Проверка "**если** `дистанция[ребро[e].A < макс_значение` **то**" нужна, только если граф содержит рёбра отрицательного веса: без такой проверки бы происходили релаксации из вершин, до которых пути ещё не нашли, и появлялись бы некорректные расстояния вида $\infty - 1$, $\infty - 2$, и т.д.

```
БеллманФорд(A):
     для v из V
          дистанция[v] = макс_значение
     дистанция[A] = 0

     пока истина
       релаксация = ложь
       для e=0 до m
         если дистанция[ребро[e].A] < макс_значение то
           если дистанция[ребро[e].B] <
                дистанция[ребро[e].A]+ребро[e].вес то
                   дистанция[ребро[e].B] =
                      дистанция[ребро[e].A]+ребро[e].вес]
                   путь[ребро[e].B] = путь[ребро[e].A]
                   путь[ребро[e].B].добавить(ребро[e])
                   релаксация = истина
       если не релаксация то
          прервать
```

На каких графах эта оптимизация будет давать выигрыш?

Отрицательные циклы

Алгоритм Беллмана-Форда позволяет проверить наличие или отсутствие цикла отрицательного веса в графе, а при его наличии — найти один из таких циклов.

Для нахождения цикла нужно Делается n итераций алгоритма Беллмана-Форда, и если на последней итерации не произошло никаких изменений — то отрицательного цикла в графе нет. В противном случае возьмем вершину, расстояние до которой изменилось, и будем идти от неё по предкам, пока не войдём в цикл; этот цикл и будет искомым отрицательным циклом.

```
текущий = путь[релаксация].Последний;
пока текущий != релаксация
  ...
  текущий = путь[релаксация].Предыдущий(текущий);
```

Алгоритм Флойда — Уоршелла

Алгоритм Флойда — Уоршелла — динамический алгоритм для нахождения кратчайших расстояний между всеми вершинами взвешенного ориентированного графа. Разработан в 1962 году Робертом Флойдом и Стивеном Уоршеллом. При этом алгоритм впервые разработал и опубликовал Бернард Рой в 1959 году.

Идея алгоритма

Вершины графа последовательно пронумерованы от 1 до **n.** Алгоритм использует матрицу D размера п * n, в кот. вычисляются длины кратчайших путей. Вначале D[i, j] = A[i, j] для всех i != j. Если дуга i -» j отсутствует, то D[i, j] = ∞. Каждый диагональный элемент матрицы D равен 0.
Над матрицей D выполняется n итераций. После k-й итерации A[i, j] содержит значение наименьшей длины путей из вершины **i** в вершину **j**, которые не проходят через вершины с номером, большим **k**. Другими словами, между концевыми вершинами пути **i** и **j** могут находиться только вершины, номера

которых меньше или равны **k**. На **k-й** итерации для вычисления матрицы А применяется следующая формула:

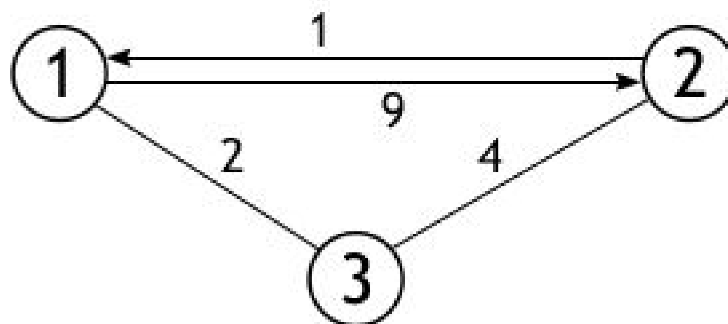
$$D_k[i,j]=\min(D_{k-1}[i,j],\ D_{k-1}[i,k]+D_{k-1}[k,j])$$

Сложность алгоритма имеет порядок 0(V^3)

Положим, что в качестве матрицы смежности, каждый элемент которой хранит вес некоторого ребра, была задана следующая матрица:

0	9	2
1	0	4
2	4	0

Количество вершин в графе, представлением которого является данная матрица, равно 3, и, причем между каждыми двумя вершинами существует ребро. Вот собственно сам этот граф:



Задача алгоритма: перезаписать данную матрицу так, чтобы каждая ячейка вместо веса ребра из i в j, содержала кратчайший путь из i в j. Для примера взят совсем маленький граф, и поэтому не будет не чего удивительного, если матрица сохранит свое изначальное состояние. Но результат тестирования программы показывает

замену двух значений в ней. Следующая схема поможет с анализом этого конкретного примера.

k	i	j	замена
1	1	1	
1	1	2	
1	1	3	
1	2	1	
1	2	2	
1	2	3	3<4, D[2][3] ←3
1	3	1	
1	3	2	
1	3	3	
2	1	1	
2	1	2	
2	1	3	
2	2	1	
2	2	2	
2	2	3	
2	3	1	
2	3	2	
2	3	3	
3	1	1	
3	1	2	6<9, D[1][2] ←6
3	1	3	
3	2	1	
3	2	2	
3	2	3	
3	3	1	
3	3	2	
3	3	3	

0	9	2
1	0	4
2	4	0

мат. смеж.

0	9	2
1	0	3
2	4	0

0	6	2
1	0	4
2	4	0

В данной таблице показаны 27 шагов выполнения основной части алгоритма. Их столько по той причине, что время выполнения метода равно O(|V|3). Наш граф имеет 3 вершины, а 3^3=27. Первая замена происходит на итерации, при которой k=1, i=2, а j=3. В тот момент D[2][1]=1, D[1][3]=2, D[2][3]=4. Условие истинно, т. е. D[1][3]+D[3][2]=3, а 3<4, следовательно, элемент матрицы D[2][3] получает новое значение. Следующий шаг, когда условие также истинно привносит изменения в элемент, расположенный на пересечении второй строки и третьего столбца.

Псевдокод

```
ФлойдУоршелл():
    для k = 1 до n
        для i = 1 до n
            для j = 1 до n
                D[i][j] = min(D[i][j], D[i][k] + D[k][j])
```

Нюансы?

```
Инициализация():
    для i = 1 до n
        для j = 1 до n
            D[i][j] = i == j ? 0 :
                A[i,j] == 0 ? макс_значение : A[i,j]
```

Восстановление путей?

```
ФлойдУоршелл():
    для k = 1 до n
        для i = 1 до n
            для j = 1 до n
                если D[i][j] < D[i][k] + D[k][j] то
                    D[i][j] = D[i][k] + D[k][j]
                    путь[i,j] = k
```

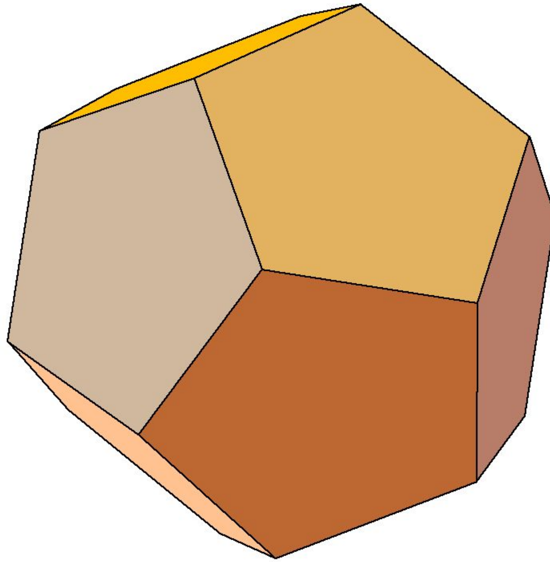
```
Путь(A, B):
     текущая = путь[A,B]
     пока текущая != A
          стек.Добавить(текущая)
          текущая = путь[A, текущая]
```

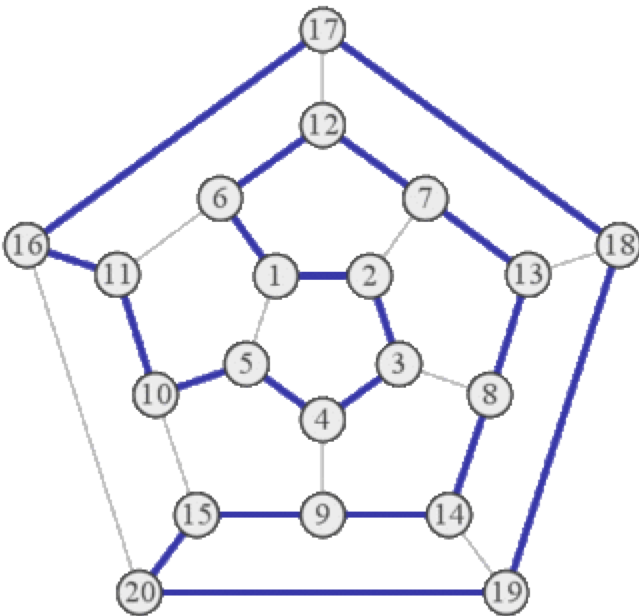
Отрицательные циклы?

Вопросы?

Гамильтонов цикл

Гамильтоновы цикл назван в честь ирландского математика У. Гамильтона, который впервые исследовав задачу «кругосветного путешествия» по додекаэдру. В этой задаче вершины додекаэдра символизировали известные города, такие как Брюссель, Амстердам, Эдинбург, Пекин, Прага, Дели, Франкфурт и др., а рёбра — соединяющие их дороги. Путешествующий должен пройти «вокруг света», найдя путь, который проходит через все вершины ровно один раз[3]. Чтобы сделать задачу более интересной, порядок прохождения городов устанавливался заранее. А чтобы было легче запомнить, какие города уже соединены, в каждую вершину додекаэдра был вбит гвоздь, и проложенный путь отмечался небольшой верёвкой, которая могла обматываться вокруг гвоздя. Однако такая конструкция оказалась слишком громоздкой, и Гамильтон предложил новый вариант игры, заменив додекаэдр плоским графом, изоморфным графу, построенному на рёбрах додекаэдра





Гамильто́нов граф — математический объект теории графов. Представляет собой граф (набор точек и соединяющих их линий), который содержит *гамильтонов цикл*.

Гамильтоновым циклом является такой цикл (замкнутый путь), который проходит через каждую вершину данного графа ровно по одному разу.

Гамильтонова пути, который является простым путём(путём без петель), проходящим через каждую вершину графа ровно один раз. Гамильтонов путь отличается от цикла тем,

что у пути начальные и конечные точки могут не совпадать, в отличие от цикла. Гамильтонов цикл является гамильтоновым путём, но не наоборот.

Необходимое условие существования гамильтонова цикла в неориентированном графе: если неориентированный граф G содержит гамильтонов цикл, тогда в нём не существует ни одной вершины v(i) с локальной степенью $p(v(i))<2$.

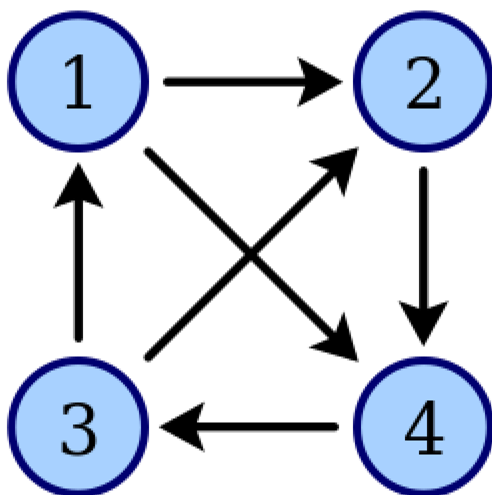
Доказательство следует из определения.

Строгого решения задачи, кроме как полным перебором не существует

Похожа на данную и ***задача о коммивояжере,*** которая тоже состоит в построении цикла, проходящего по всем городам по одному разу, но при этом требуется минимизировать транспортные расходы. Алгоритма решения данной задачи тоже не существует

Турнир — это ориентированный граф, полученный из неориентированного полного графа путём назначения направления каждому ребру. Таким образом, турнир — это орграф, в котором каждая пара вершин соединена одной направленной дугой.

Много важных свойств турниров рассмотрены Ландау (*Landau*) для того, чтобы исследовать модель доминирования цыплят в стае. Текущие приложения турниров включают исследования в области голосования и коллективного выбора среди других прочих вещей. Имя *турнир* исходит из графической интерпретации исходов кругового турнира, в котором каждый игрок встречается в схватке с каждым другим игроком ровно раз, и в котором не может быть ничьих. В орграфе турнира вершины соответствуют игрокам. Дуга между каждой парой игроков ориентирована от выигравшего к проигравшему. Если игрок **a** побеждает игрока **b**, то говорят, что **a** ***доминирует над*** **b**.

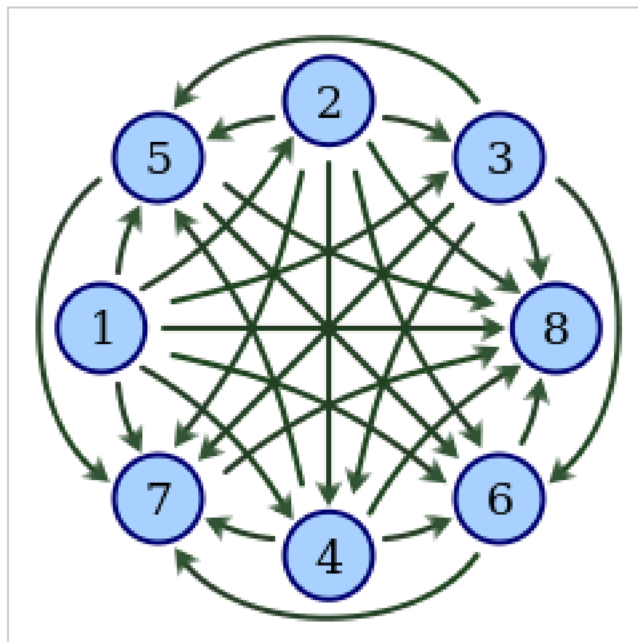


Транзитивность

Турнир, в котором (a -> b) и (b -> c) => (a -> c), называется **транзитивным**. В транзитивном турнире вершины могут быть полностью упорядочены в порядке достижимости.

Следующие утверждения для турнира с *n* вершинами эквивалентны:

1. *T* транзитивен.
2. *T* ацикличен.
3. *T* не содержит циклов длины 3.
4. Последовательность числа выигрышей (множество полуисходов) *T* есть $\{0,1,2,\ldots,n-1\}$.
5. *T* содержит ровно один гамильтонов путь



Теория Рамсея — раздел математики, изучающий условия, при которых в произвольно формируемых математических объектах обязан появиться некоторый порядок. Названа в честь Фрэнка Рамсея.

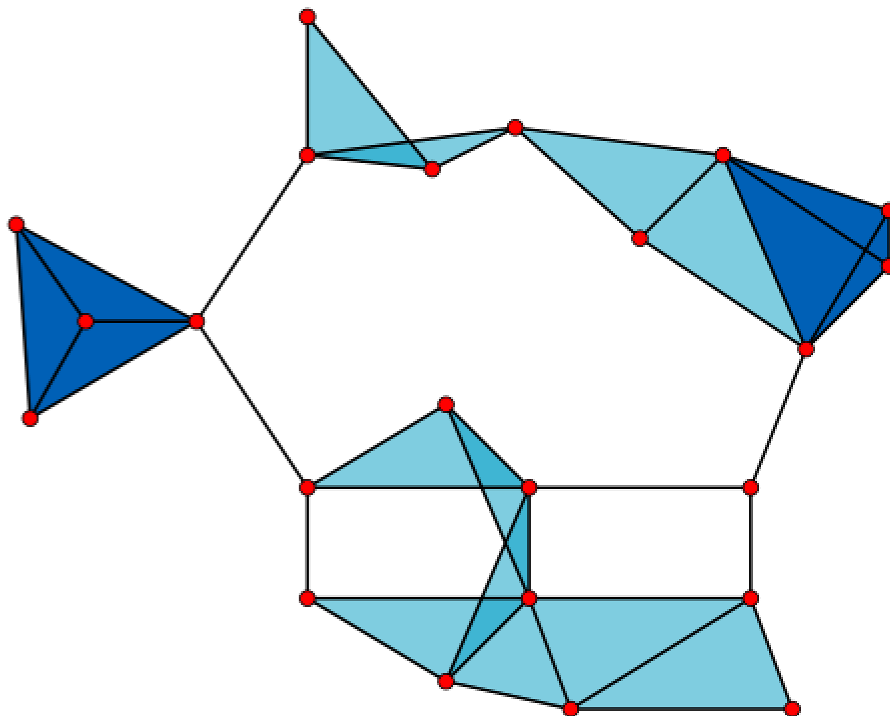
Задачи в теории Рамсея обычно звучат в форме вопроса «сколько элементов должно быть в некотором объекте, чтобы гарантированно выполнялось заданное условие или существовала заданная структура». Простейший пример:

- *Доказать, что в любой группе из 6 человек, найдутся либо три человека, знакомые друг с другом, либо три человека, попарно незнакомые друг с другом.*

Транзитивные турниры играют существенную роль в теории Рамсея, аналогичную роли, которую играют клики в неориентированных графах.

Кликой неориентированного графа называется подмножество его вершин, любые две из которых соединены ребром.

Хотя изучение полных подграфов началось ещё с формулировки теоремы Рамсея в терминах теории графов Эрдёшем и Секерешем. Термин «**клика**» пришёл из работы Люка и Пери, использовавших полные подграфы при изучении социальных сетей для моделировании клик людей, то есть групп людей, знакомых друг с другом. Клики имеют много других приложений в науке, и, в частности, в биоинформатике.



```
DSF(v)
посетили[v] = истина;
путь.Добавить(v);
для u из v
    если не посетили[u] то
        если DFS(u) то
            вернуть истина
если НетНеПосещенныхВершин() то
    вернуть истина
посетили[v] = ложь
путь.УдалитьПоследний()
вернуть ложь
```

Гамильтонов цикл?

```
ГамильтоновПуть():
    для v из V
        если DFS(v) то
            вернуть истина
    вернуть ложь
```