Analysis of the minimum spanning tree problem
5 semester

January 2023

Автор:
Клячин Артемий

Содержание

Аннотация

В работе рассматривается задача поиска такого остовного дерева в графе $G = (V, E)$, что максимальная степень вершин дерева (далее степень дерева) минимальна. Приведено доказательство факта, что проверка существования остовного дерева степени не выше k является NP-полной для любого фиксированного $k \geqslant 2$. Представлен алгоритм поиска аппроксимированного остовного дерева, степени не более чем на единицу превышающего степень оптимального дерева, работающий за полиномиальное время.

Работа состоит из двух частей: теоретической и практической. Теоретическая часть работы включает в себя частичное изложением статьи Fürer и Raghavachari [7] 1994 года. Переведены две леммы, две теоремы, идеи алгоритма, сам алгоритм, доказательство оценки времени работы. Мной добавлен ряд уточнений в доказательства утверждений, тем самым они стали более подробными и понятными. Пошаговый алгоритм был изменён по сравнению с оригиналом для удобства реализации на языке Python.

1 Введение

Работа посвящена задаче поиска остовного дерева минимальной степени. Данная проблема интересует научное сообщество с точки зрения, того, что она является NP-полной. Если удастся доказать, что существует NP-полная задача, которую можно решить за полиномиальное время, то будет доказана одна из главных математических гипотез - равенство $P = NP$. Понятие NP-полноты также относится к нашей задаче, этот факт будет доказан ниже. Эта задача имеется в классификаторе Garey и Johnson [1, 2] под обозначением [ПС1] "Остов ограниченной степени"([ND1] "Degree Constrained Spanning Tree").

Задача поиска минимального остовного дерева (MDST) имеет практическое применение. Как у многих задач на графы она применяется в сетевых системах. На данный момент находить оптимальное остовное дерево мы умеет только за экспоненциальное время. Если граф имеет размер более 1000 вершин, то это неподъёмная задача для имеющихся вычислительных мощностей. Решением этой ситуации может стать нахождение аппроксимированного остовного дерева, степени не более чем на единицу превышающего степень оптимального дерева.

Основные понятия используемые в статье: *остовное дерево графа, степень остовного дерева, минимальное (оптимальное) остовное дерево графа (MSDT), NP-полнота.*

1.1 История вопроса

В 1981 Коу [4] предложил алгоритм поиска аппроксимированного остового дерева для задачи Штейнера, которая состоит в поиске кратчайшей сети, соединяющей заданный конечный набор точек плоскости (названа в честь швейцарского математика Якоба Штейнера). Позже в 1992 году Fürer и Raghavachari [6] построили алгоритм аппроксимированного решения MSDT. Понятие NP-полноты было введено в 1971 г.

1.2 Важность проблемы

Любая задача, в которой надо составить граф с ограниченным числом выходящих из вершины рёбер, является частным случаем задачи MSDT. Например, если требуется построить сеть передачи данных без резервирования, использую при этом марштутизаторы с минимальным числом портов, тем самым минимизируя бюджет на построения сети. Мы можем решить эту задачу с помощью поиска минимального остового дерева.

1.3 Связь с близкими результатами

1.3.1 Задача Штейнера на графах

Задача Штейнера является обобщением задачи поиска минимального остовного дерева. Она имеет больше практических приложений. Эта задача наряду с входным графом, получает множество терминальных множество вершин D. Необходимо выделить в графе поддерево минимальной степени (дерево Штейнера), содержащее множество D. MDST - это задача Штейнера, в которой D является множеством всех вершин. Доказательство NP-полноты задачи Штейнера является аналогичным NP-полноты MSDT. Также существует работающий за полиномиальное время алгоритм поиска аппроксимированного дерева Штейнера, степени не более чем на единицу превышающего степень оптимального дерева Штейнера.

2 Основные понятия и утверждения

Понятия

1) *Остовное дерево графа* G - это дерево, подграф G, с тем же множеством вершин, что и G.

2) *Степень остовного дерева* - это наибольшее значение из степеней всех вершин.

3) Обозначим Δ_G минимальную степень остовного дерева графа G из все возможных остовных деревьев G.

4) *Минимальное (оптимальное) остовное дерево графа* G - это остовное дерево графа G степени Δ_G.

5) MDST - задача поиска минимального остовного дерева

Доказательство NP-полноты задачи:

Утверждение *Задача проверки существования остовного дерева в графе G степени не выше k принадлежит классу NP для любого фиксированного $k \geqslant 2$.*

Утверждение *Задача проверки существования остовного дерева в графе G степени не выше k является NP-полной для любого фиксированного $k \geqslant 2$.*

Полиномиальный алгоритм для поиска аппроксимированного дерева:

Лемма 1. Пусть W - свидетельствующее множество размера w, удаление которого разбивает G на t компонент связности. Тогда $\Delta_G \geqslant (w + t - 1)/w$.

Теорема 1. Дан граф G, $\tau = \inf_{X \subset V} |X|/c(X)$ - его жёсткость. Тогда $\Delta_G - 3 < 1/\tau \leqslant \Delta_G$.

Теорема 2. Пусть $G = (V, E)$ - заданный граф. Пусть T - остовное дерево G степени k. Пусть S - множество вершин степени k в T. Пусть B - произвольное подмножество вершин степени $k-1$ в T. Пусть $S \cup B$ будет удален из графа, разбив дерево T на лес F. Предположим, что G удовлетворяет условию, что нет рёбер G между деревьями в F. Тогда $k \leqslant \Delta_G + 1$.

Лемма 2. Когда алгоритм останавливается, $k \leqslant \Delta_G + 1$.

Доказательство полиномиального времени работы алгоритма.

Вспомогательный утверждения:

τ - жёсткость графа G. k - степень графа G. $\tau \geqslant 1/(k-2)$, при $k \geqslant 3$, то существует остовное дерево графа G степени не более k [5].

Время работы алгоритма DSU ("Система непересекающихся множеств").

3 Доказательства утверждений

3.1 Доказательство NP-полноты задачи

Утверждение *Задача проверки существования остовного дерева в графе G степени не выше k является NP-полной для любого фиксированного $k \geqslant 2$.*

Доказательство. Формально, задача состоит в том, чтобы проверить принадлежность пары (G, k) к языку LIMITSPANTREE состоящему из всех пар (G', k'), которые удовлетворяют условию: существует остовное дерево в графе G' степени не выше k'. Для доказательства утверждения необходимо проверить, что LIMITSPANTREE $\in NP$. Построим недетерминированную машину Тюринга (НДМТ). В качестве сертификата будем передавать множество рёбер дерева. НДМТ проверяет, что переданный граф корректный, т.е. сертификат это список рёбер (пар вершин), любое ребро содержится в G, любые две пары вершин связны (граф связен), количество рёбер на единицу меньше числа вершин (граф является деревом), степень каждой вершины не выше k. Каждую из этих проверок можно произвести за полиномиальное время от размера графа, следовательно, за полиномиальное время от длины входа (G, k). Таким образом если $(G, k) \in$ LIMITSPANTREE, то найдётся сертификат на котором предикат выведет $True$. В другую сторону: если для некоторого сертификата предикат выводит $True$, то сертификат задаёт остовное дерево графа G со степенью не выше k, поэтому $(G, k) \in$ LIMITSPANTREE. Следовательно, LIMITSPANTREE $\in NP$.

Утверждение *Задача проверки существования остовного дерева в графе G степени не выше k является NP-полной для любого фиксированного $k \geqslant 2$.*

Доказательство. Задача поиска гамильтонового пути в графе является NP-полной. Это частный случай задачи MDST при $k = 2$. Следовательно, MDST является NP-полной. Если рассматривать задачу для заданного $k > 2$, то можно показать NP-полноту задачи для подмножества графов. Так можно взять некоторый связный граф, добавить к каждой вершине по $k - 2$ листа (ребро с вершиной). Размер графа увеличился не более чем полиномиально. А задача поиска остовного дерева степени не выше k на нём, сведётся к задаче поиска гамильтонового пути на начальном графе.

3.2 Полиномиальный алгоритм для поиска аппроксимированного дерева

Определим следующее понятие *жёсткости графа* G (понятие *graph toughness* было введено в [3]). Пусть $X \subset V$ - произвольное подмножество вершин, и пусть граф G без вершин X содержит $c(X)$ компонент связности. *Жёсткость* графа G, обозначаемая τ, определяется как минимальное отношение $|X|/c(X)$ по всем подмножествам X из V при условии $c(X) > 1$. То есть жесткость графа - это минимальное отношение, генерируемое подмножеством X, которое дает наибольшее

количество компонент связности на удаленную вершину.

Пусть $W \subset V$ - подмножество вершин размера w. Предположим, что удаление W из G разъединяет G на t компонент. Пусть $d = (w + t - 1)/w$. Мы говорим, что множество W является *свидетельствующим*, так как оказывается, что $\Delta_G \geqslant d$. Мы сформулируем это как лемму.

Лемма 1. Пусть W - свидетельствующее множество размера w, удаление которого разбивает G на t компонент. Тогда $\Delta_G \geqslant (w + t - 1)/w$.

Доказательство. Вершины G сгруппируем в подмножества следующим образом. Каждая вершина $u \in W$ образует свой собственный одноэлементный набор. Остальные вершины сгруппированы в t подмножеств, соответствующих t компонентам связности в $G\backslash W$. Мы определили $w + t$ непересекающихся подмножеств V, и каждое остовное дерево из G, содержит по крайней мере $w + t - 1$ ребра, соединяющие эти подмножества, и все эти ребра инцидентны по крайней мере одной вершине в W. Следовательно, средняя степень вершин из W в любом остовном дереве не меньше $(w + t - 1)/w$ и в W есть некоторая вершина, степень которой не меньше, по крайней мере, средней степени. Следовательно, в любом остовном дереве степень некоторой вершины в W не менее $(w+t-1)/w$. Поскольку Δ_G является степенью некоторого дерева остовного дерева, мы получаем $\Delta_G \geqslant (w + t - 1)/w$.

Теорема 1. Дан граф G, $\tau = \inf_{X \subset V} |X|/c(X)$ - его жёсткость. Тогда $\Delta_G - 3 < 1/\tau \leqslant \Delta_G$.

Доказательство теоремы 1. Если $\Delta_G \leqslant 3$, то первое неравенство очевидно. Если $\Delta_G \geqslant 4$: Предположим противное. $\tau \geqslant 1/(\Delta_G - 3)$ для $\Delta_G \geqslant 4$. Мы можем применить результат Win[4] о том, что если $\tau \geqslant 1/(k - 2)$, при $k \geqslant 3$, то существует остовное дерево графа G степени не более k. Тогда G имеет остовное дерево степени не более $\Delta_G - 1$. Это противоречие, потому что, по определению, G не имеет остовного дерева со степенью меньше Δ_G. Следовательно, $\tau < 1/(\Delta_G-3)$, что эквивалентно первому неравенству в формулировке теоремы.

Пусть $X \subset V$ - подмножество вершин, достигающее коэффициента жёсткости τ. Пусть $c(X)$ - количество компонент связности, сгенерированных при удалении X из G. По определению, $\tau = |X|/c(X)$. Мы применяем лемму 1, где X является свидетельствующим множеством W в формулировке леммы и выводим, что $\Delta_G \geqslant (|X|+c(X)-1)/|X|$. Следовательно, $\Delta_G \geqslant 1+1/\tau-1/|X| \geqslant 1/\tau$. Это эквивалентно второму неравенству в формулировке теоремы.

Теорема 2. Пусть $G = (V, E)$ - заданный граф. Пусть T - остовное дерево G степени k. Пусть S - множество вершин степени k в T. Пусть B - произвольное подмножество вершин степени $k-1$ в T. Пусть $S \cup B$ будет удален из графа, разбив дерево T на лес F. Предположим, что G удовлетворяет условию, что нет рёбер G между деревьями в F. Тогда $k \leqslant \Delta_G + 1$.

Доказательство. Сумма степеней вершин $S\cup B$ в графе T равна $|S|k+|B|(k-1)$. Поскольку T является ациклическим, существует не более $|S\cup B|-1$ ребер, оба конца которых в $S\cup B$. Следовательно, число ребер, инцидентных вершинам $S\cup B$, не менее $d = |S|k+|B|(k-1)-(|S\cup B|-1)$. Лес, полученный из T путем удаления этих ребер, имеет не менее $d+1$ компонент связности. Вершины $S\cup B$ будут представлены отдельными компонентами. Убрав их из леса получим лес F. Получаем, что лес F содержит $d + 1 - |S \cup B|$ компонент связности, это равно $|S|(k - 2) + |B|(k - 3) + 2$. Применим лемму 1 для графа G и свидетельствующего множества $S\cup B$. Поскольку лемма применяется на G необходимо, чтобы между деревьями F, не было рёбер из G, что требуется в условии теоремы. Тогда $w = |S|+|B|$, $t \geqslant |S|(k-2)+|B|(k-3)+2$, а результат леммы $\Delta_G \geqslant (w+t-1)/w$, будет иметь вид $\Delta_G \geqslant (|S|(k-1)+|B|(k-2)+1)/(|S|+|B|) > k-2$. Следовательно, $k \leqslant \Delta_G + 1$.

Замечание. Условие теоремы 2, может выполняться или не выполняться в зависимости от

выбора подмножества B вершин степени $k-1$ в T. Потому что от этого зависит есть ли рёбра G между деревьями F или нет.

Идея алгоритма. Давайте сначала изучим более простой случай построения связующего дерева. Мы используем следующую стратегию, алгоритм начинается с произвольного остовного дерева T из G и пытается уменьшить его степень. Максимальная степень T всегда обозначается через k. Пусть $p(u)$ обозначает степень вершины u в T. Мы определяем, "улучшение"основанное на некоторых локальных свойствах графа, как базовый строительный блок для уменьшения максимальной степени. Алгоритм продолжает вносить улучшения до тех пор, пока может это делать. Потом алгоритм завершается и выводит остовное дерево. Алгоритм пытается сформировать **свидетельствующий набор** с вершинами степени k. Теперь мы определяем понятие улучшения, которое мы используем в качестве строительного блока нашего алгоритма.

Определение 1. Пусть (u, v) - ребро G, которого нет в T. Пусть C - цикл, генерируемый при добавлении (u, v) к T. Заметим, что такой цикл единственный. Предположим, что в C есть вершина w степени k, в то время как степени вершин u и v не более $k-2$. Улучшение T - это модификация T путем добавления ребра (u, v) к T и удаления одного из ребер в C, инцидентного w. При таком улучшении мы говорим, что w **извлекает выгоду из** (u, v).

Обратите внимание, что мы назвали этот шаг улучшением, потому что количество вершин максимальной степени уменьшилось на единицу. Приведенную выше идею можно использовать для многократного уменьшения числа вершин максимальной степени. Единственный случай, который препятствует прогрессу, возникает, когда либо u, либо v или оба имеют степень $k-1$. Мы изучим этот случай более подробно.

Определение 2. Пусть T - остовное дерево степени k. Пусть $p(u)$ обозначает степень вершины в дереве T, $u \in T$. Пусть $(u, v) \notin T$ - ребро в G. Предположим, что w - вершина степени k в цикле, сгенерированном добавлением (u, v) к T. Если $p(u) \geqslant k-1$, мы говорим, что u **блокирует** w **из** (u, v).

3.3 Алгоритм $\Delta_G + 1$.

Алгоритм реализуется по принципу "снизу вверх"следующим образом. Алгоритм на каждой итерации либо уменьшает множество S_k, либо определяет, что дерево аппроксимировано. Для того, чтобы находи итоговое свидетельствующее множество (из теоремы 2) вводим понятие плохих и хороших вершин. В начале каждой итерации алгоритма все вершины в $S_k \cup S_{k-1}$ удаляются из T и помечаются как плохие. Все остальные вершины помечены как хорошие. В будущем некоторые вершины степени $k-1$ будут помечены как хорошие. Рассмотрим компоненты, образованные ребрами дерева проведёнными между хорошими вершинами. Если в какой-нибудь момент между хорошими компонентами нет ребер, алгоритм останавливается. В этом случае по теореме 2, множество оставшихся плохих вершин является свидетельствующим множеством, а остовное дерево удовлетворяет $k \leqslant \Delta_G + 1$. В противном случае, пусть (u, v) будет ребром между двумя хорошими компонентами F_u и F_v. Степень вершин u и v не более $k-1$. Мы рассматриваем возможность добавления (u, v) к T, которое создаст цикл. Если в этом цикле нет вершин степени k, то есть по крайней мере одна плохая вершина степени $k-1$. Тогда мы помечаем все плохие вершины в этом цикле (это вершины степени $k-1$) как хорошие и объединяем все компоненты в этом цикле вместе со всеми его вершинами степени $k-1$ в одну хорошую компоненту связности. Когда мы помечаем вершину как хорошую, то запоминаем пару вершин (u, v). С помощью этой пары мы сможем в будущем уменьшить степень одной из вершин степени $k-1$ в этом цикле.

В противном случае, в цикле есть вершины степени k, то выбираем любую. Обозначим её за w.

Начинаем процесс, который приведёт к уменьшению степени этой вершины, после мы перейдём к следующей итерации. Если текущие вершины u и v неблокирующие (т.е. их степень не больше $k-2$), то проводим улучшение описанной в определении 1. Иначе хотя бы одна из вершин блокирующая (её степень $k-1$). Чтобы провести действие описанное в определении 1 необходимо уменьшить степень блокирующей вершины на 1. Это мы делаем также по описанному здесь методу, но для вершин u' и v', которые мы запомнили, когда сделали эту вершину хорошей. За w берём блокирующую вершину. Данный алгоритм рекурсивный. Как только мы сможем произвести некоторое изменение T, проводим цепочку изменений, которым было необходимо это изменение и так далее.

Остаётся вопрос, почему эти улучшения не станут мешать друг другу и когда-нибудь остановятся. На каждой подытерации мы рассматриваем некоторое w, степень которого нам надо уменьшить на 1, а также его u и v. Назовём K_u и K_v - деревья из леса $T\backslash\{w\}$) содержащие u и v соответственно. В дереве T вершина w находится между u и v, поэтому $K_u \neq K_v$. Если вершина u является блокирующей, то её вершины u' и v' будят находится в K_u. Поскольку иначе вершина w стала бы хорошей ещё на шаге (u', v'). Следовательно, подытерация для $w = u$, изменит только K_u. Если вершина v блокирующая, то аналогично только K_v. Для последовательности блокирующих вершин в нашем алгоритме деревья K будут вложены друг в друга. Поэтому блокирующие вершины не будут мешать друг другу. Алгоритм остановится, поскольку, для каждой последующей блокирующей вершины деревья K, будет всё меньше и меньше. Так не может происходить вечно. В какой-нибудь момент найдётся блокирующая вершина, которую можно улучшить с помощью двух неблокирующих.

Если в этом цикле есть вершина степени k, мы можем применить улучшение . Внесение этого изменения уменьшит размер S_k. В противном случае в цикле нет вершин степени k, но есть по крайней мере одна плохая вершина степени $k-1$. Тогда мы помечаем все плохие вершины в этом цикле как хорошие и объединяем все компоненты в этом цикле вместе со всеми его вершинами степени $k-1$ в одну хорошую компоненту связности. Запоминаем это ребро (u, v) и этот цикл, как возможное улучшение в будущем в случае, если в нём появится вершина степени k. Затем мы возвращаемся к поиску других ребер между хорошими компонентами. Если попалось ребро инцидентное вершине степени $k-1$, то оно называется блокирующим. Во всех случаях мы либо находим способ уменьшить степень некоторой вершины в S_k, либо находим блокирующее множество, позволяющее нам применить теорему 2. Мы покажем, что количество итераций ограничено $O(nlogn)$. Мы также покажем, что каждая фаза алгоритма может быть реализована в полиномиальное время. Объединяя эти факты, мы получаем алгоритм, который удовлетворяет теореме 1.2. На рисунке 1 приведен алгоритм, реализующий

3.4 Пошаговый алгоритм

Входные данные: граф G.
Результат: Остовное дерево T, которое аппроксимирует MDST.

1. Найдите остовное дерево T из G.
2. Выполняйте следующие действия до тех пор, пока алгоритм не остановится.
   (a) Пусть k - его степень. Отметьте вершины степени k и $k-1$ как плохие, остальные как хорошие. Удалите плохие вершины из T, получите лес F
   (b) До тех пор, пока существует ребро (u, v) в G, соединяющее две разные компоненты связности F и все вершины степени k помечены как плохие, выполняем действия:
      i. Найдите плохие вершины в цикле C, порожденном T вместе с (u, v), и отметьте их как хорошие. Для каждой такой вершины запомните (u, v), при которых они стали хорошими.

ii. Обновите F путем объединения компонент связности в цикле C, эти вновь отмеченные вершины объединяются в единый компонент. Обратите внимание, что на этом этапе более двух компонент F могут быть объединены в один.

iii. Если есть вершина w степени k, отмеченная как хорошая, выполняем функцию улучшения для w (это последовательность улучшений, которые распространяются на w и обновляют T). После чего останавливаем работу цикла (b).

Функцию улучшения для вершины w'. Эта функция работает для хорошей вершины степени k или $k-1$. (u', v') - пара, благодаря которой она стала хорошей. Выполняем следующие действий.

A. Если степень u' и v' в остовном дереве не больше $k-2$, тогда добавляем в остовное дерево ребра (u', v') и удаляем ребро инцидентное w.

B. Если степень u' в остовном дереве равна $k-1$, тогда рекурсивно выполняем улучшение для вершины u'. После возврата из рекурсии добавляем в T ребро (u', v'), удаляем ребро инцидентное w'.

C. Если степень v' в остовном дереве равна $k-1$, делаем аналогичное действие.

(c) Если в предыдущем пункте не было выполнено действие (iii), то в дереве не осталось вершин степени k, которые можно улучшить. Останавливаем работу цикла (2).

3. Выводите конечное дерево T, его степень k и свидетельствующее множество W, состоящий из всех вершин помеченных как плохие.

конец

Лемма 2. Когда алгоритм останавливается, $k \leqslant \Delta_G + 1$.

Доказательство. Пусть S это S_k, а B - плохие вершины степени $k-1$. Обратите внимание, что алгоритм останавливается только тогда, когда между хорошими компонентами нет ребер. Следовательно, дерево T вместе с этими множествами S и B удовлетворяет условиям теоремы 2, и мы получаем желаемый результат.

3.5 Доказательство полиномиального времени работы алгоритма.

В начале покажем, что количество итераций работы алгоритма полиномиально. Сумма степеней вершин дерева в точности равна $2(n-1)$. Следовательно, число вершин степени k в дереве на n вершинах равно $O(n/k)$. Поскольку размер S уменьшается на единицу в каждой фазе (кроме последней), существует $O(n/k)$ фаз, когда максимальная степень равна k. Суммируя гармонический ряд, соответствующий различным значениям k, мы приходим к выводу, что существует $O(n \log n)$ фаз. На каждом этапе мы пытаемся найти улучшения, которые распространяются на вершины S_k. Лемма 4.1 (её пока нет, может без неё) гарантирует, что всякий раз, когда мы находим вершину w степени k, можно найти последовательность улучшений, которая распространяется на w. Лемма 2 показывает, что когда алгоритм останавливается, степень результирующего дерева находится в пределах единицы от оптимальной степени. Каждая фаза алгоритма может быть реализована почти за линейное время с использованием быстрого алгоритма объединения непересекающихся множеств Тарьяна для поддержания компонент связности (например, см. [8]). Следовательно, весь алгоритм выполняется в $O(mn\alpha(m,n)\log n)$, где m - количество ребер, а α - обратная функция Аккермана.

3.6 Практическая часть.

3.6.1 Реализация алгоритма на языке Python $\Delta_G + 1$

Для реализации алгоритма $\Delta_G + 1$ воспользуемся следующими библиотеками Python:

- NetworkX - для представления графов, а также для генерации разных графов
- DisjointSet - реализация “Disjoint Set Union”
- collection.queue - реализация FIFO-queue * NumPy - для генерации случайных чисел
- Matplotlib - для визуализации графов

[1]:
```
import networkx as nx
from disjoint_set import DisjointSet
from collections import deque
import numpy as np
import matplotlib.pyplot as plt
```

[2]:
```
def plot_tree_over_graph(T, G=None, pos=None, title=None):
    """Plot the tree T of the graph G with Matplotlib.

    Parameters
    ----------
    T : tree
        A networkx graph

    G : graph
        A networkx graph, optional

    pos :  dictionary, optional
        A dictionary with nodes as keys and positions as values.
        If not specified a spring layout positioning will be computed.
        See :py:mod:`networkx.drawing.layout` for functions that
        compute node positions.

    title : string, optional
        The title for the plot
    """
    plt.figure(figsize=(4, 3))
    if pos is None:
        pos = nx.drawing.spring_layout(G if G is not None else T)
    if G is not None:
        nx.draw_networkx(G, pos, edge_color='grey', node_color='white',
                         style=':')
    nx.draw_networkx(T, pos, edge_color='red', node_color='red')
    if title:
        plt.title(title)
    plt.show()

def near_minimum_degree_spanning_tree(G, T=None, seed=None, pos=None,
```

```
    trace=False, plot_initial=False, plot_final=False, plot_all=False,
    plot_iterations=[]):
    """Near-optimal Minimum Degree Spanning Tree algoritms

    Parameters
    ----------
    G : graph
        A networkx graph, optional

    T : starting tree to iterate, optional
        A networkx graph
        If tree is absent, DFS-tree from the first vertex will be taking
        examples of possible tree generation:
        nx.dfs_tree(G, next(iter(G.nodes))).to_undirected()
        nx.bfs_tree(G, next(iter(G.nodes))).to_undirected()
        nx.minimum_spanning_tree(G, algorithm='boruvka')


    seed : int or None, optional (default=None)
        Set the random state for algorithm edge selection and for plots.

    pos : dictionary, optional
        A dictionary with nodes as keys and positions as values.
        If not specified a spring layout positioning will be computed.
        See :py:mod:`networkx.drawing.layout` for functions that
        compute node positions.

    trace : boolean, optional (default=False)
        Print information about iterations

    plot_initial : boolean, optional (default=False)
        Plot the initial tree

    plot_final : boolean, optional (default=False)
        Plot the final tree

    plot_all : boolean, optional (default=False)
        Plot trees after all iterations

    plot_iterations : list of ints, optional
        Plot trees only after specified iterations

    Returns
    -------
    T : near-optimal tree

    k : maximum degree of build tree T
```

```
    W : witness set of nodes (to prove T is close to optimal)
    """

    if seed is not None:
        np.random.seed(seed)
    if pos is None:
        pos = nx.drawing.spring_layout(G)

    # Шаг 1. Найдём остовное дерево T из G.

    if T is None:
        T = nx.dfs_tree(G, next(iter(G.nodes))).to_undirected()
    if plot_initial:
        plot_tree_over_graph(T, G, pos, f'Initial tree')

    # Шаг 2a. Найдём k - степень T. Вершины степени k и k - 1 как плохие
    # (поместим в множество W).
    # Для остальных вершин построим "Disjoint Set Union" F.

    iteration = 0  # номер итерации
    while True:
        iteration += 1
        # TD - DegreeView, changing authomatically with T
        TD = T.degree
        # k - максимальная степень вершин T
        k = max(d for n, d in TD)
        # W - множество плохих вершин степеней k и k - 1
        W = {n for n, d in TD if d == k or d == k - 1}

        if trace:
            print(f'Iteration: {iteration} k: {k}, W: {W}')
        # F - DSU для T-W
        F = DisjointSet()
        for u, v in T.edges(data=False):
            if u not in W and v not in W:
                F.union(u, v)

        # Шаг 2b. До тех пор, пока существует ребро (u, v) в G, соединяющее
        # две разные компоненты связности F и все вершины степени k помечены
        # как плохие, выполняем действия:
        # i. Найдём плохие вершины в цикле C, порожденном T вместе с (u, v),
        # и отметим их как хорошие. Для каждой такой вершины запомним (u, v),
        # при которых они стали хорошими.
        # ii. Обновим F путем объединения компонентов связности в цикле C,
        # эти вновь отмеченные вершины объединятся в единую компоненту.
```

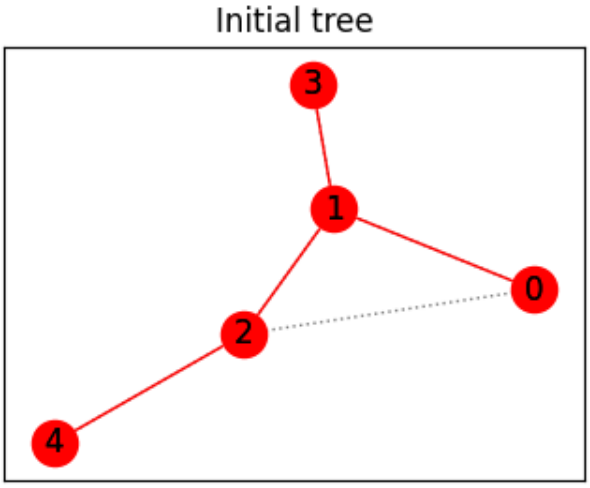
```
        improved = False  # флаг, что была улучшена вершина степени k
        candidate = None  # индекс улучшенной вершины
        unblock = {}  # {w:(u,v)} ребра (u,v), улучшающие вершину w

        E = deque([(u, v) for u, v in G.edges(data=False)
                   if u not in W and v not in W and not F.connected(u, v)])
        # E - начальный список ребёр между компонентами связности,
        #     в который могут добаляться новые рёбра по мере того,
        #     как вершины помечаются хорошими
        while E:
            u, v = E.popleft()

            if F.connected(u, v):  # u и v уже в одной комноненте связности
                continue

            C = nx.shortest_path(T, u, v)
            # C - цикл, образованный добавлением ребра (u,v) к дереву T

            candidates = [w for w in C if TD[w] == k]  # вершины степени k в C
            if candidates:
                # среди вершин степени k в цикле C случайно выбираем одну
                candidate = candidates[np.random.randint(len(candidates))]
                # запоминаем улучшающее ребро (u,v) для вершины candidate
                unblock[candidate] = (u, v)
                break

            for w in C:
                if TD[w] == k - 1 and w in W:  # плохая вершина степени k-1 в C
                    W.remove(w)  # помечаем вершину степени k-1, как хорошую
                    # дополнительные улучшающие ребра
                    E.extend([(u, v) for u, v in G.edges(w)
                              if u not in W and v not in W
                              and not F.connected(u, v)])
                    # запоминаем улучшающее ребро (u,v) для вершины w
                    unblock[w] = (u, v)
                    F.union(u, w)  # связываем компоненты u, v, w
                    F.union(v, w)

        # iii. Если есть вершина w (candidate) степени k, отмеченная
        # как хорошая, выполняем функцию улучшения для w.

        if candidate:
            Q = deque([candidate])  # FIFO очередь вершин для улучшений
            while Q:
                w = Q.popleft()
                u, v = unblock[w]
                C = nx.shortest_path(T, u, v)
```

```
                i = C.index(w)
                # случайно выбираем одно из двух рёбер вершины w цикла C
                # для удаления
                rnd = np.random.randint(2)
                edge_to_remove = (C[i - 1 + rnd], C[i + rnd])
                T.add_edge(u, v)
                T.remove_edge(edge_to_remove[0], edge_to_remove[1])
                if trace:
                    print(f'  replace the edge {edge_to_remove[0]}-'
                          f'{edge_to_remove[1]} with an edge {u}-{v}')

                if TD[u] == k:  # если вершина u стала блокирующей
                    Q.append(u)
                if TD[v] == k:  # если вершина v стала блокирующей
                    Q.append(v)

            if plot_all or iteration in plot_iterations:
                plot_tree_over_graph(T, G, pos,
                                     f'Iteration {iteration}, improved tree')

        # Шаг 2c. Если в предыдущем пункте не было выполнено действие (iii),
        # то в дереве не осталось вершин степени k, которые можно улучшить.
        # Останавливаем работу алгоритма.

        else:
            break

    # Шаг 3. Выводим конечное дерево T, его степень k и свидетельствующее
    # множество W, состоящий из всех вершин помеченных как плохие.

    if trace:
        print(f'Algorithm finished with k: {k}, W: {W}')
    if plot_final:
        plot_tree_over_graph(T, G, pos, f'Final tree')
    return T, k, W
```

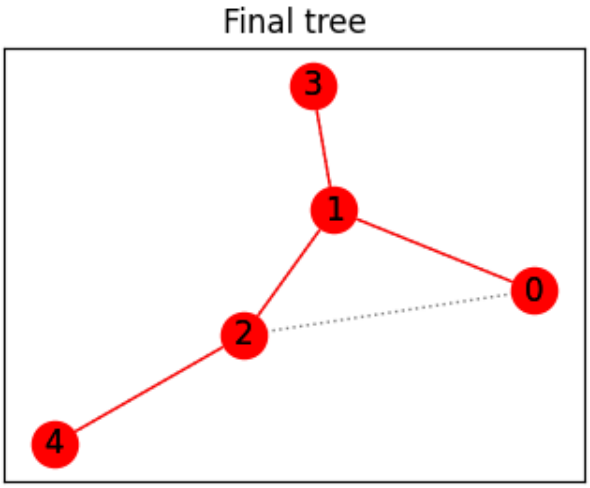
3.6.2 Пример, когда алгоритм не находит минимальную степень

Для приведённого графа, очевидно, существует гамильтонов путь, однако алгоритм не может совершить ни одной итерации и возвращает дерево со степенью $k = 3$.

[3]:
```
G = nx.bull_graph()
T, k, W = near_minimum_degree_spanning_tree(
    G, seed=10, trace=True, plot_initial=True, plot_final=True)
```



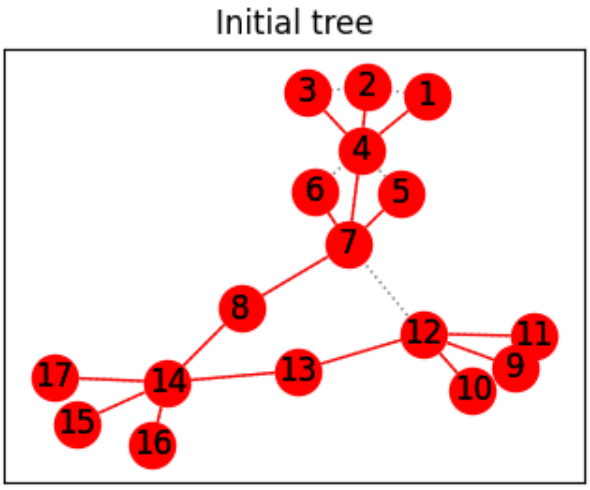

```
Iteration: 1 k: 3, W: {1, 2}
Algorithm finished with k: 3, W: {1, 2}
```



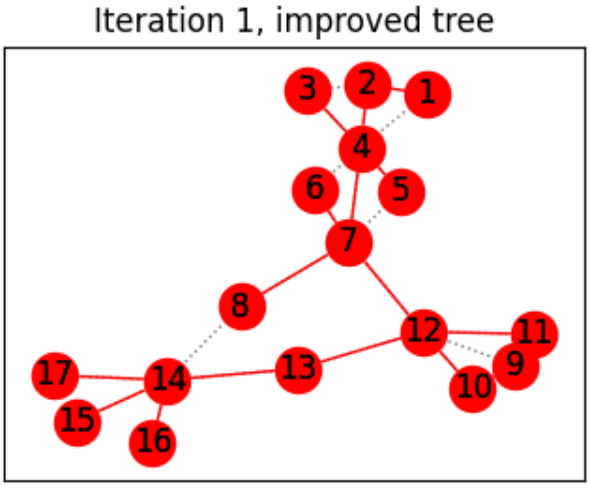
3.6.3 Пример сложной итерации

Приведём пример графа, когда на одной итерации происходит замена нескольких ребер. Перед первой итерацией множетсво плохих вершит включает $\{4, 7, 12, 14\}$. Первые три вершины имеют степень $k-1=4$, последняя степень $k=5$. Сначала мы находим, что ребро 1-2 создаёт цикл через вершину 4 и мы помечает её хорошей. Ребро 4-5 делает вершину 7 хорошей, а ребро 9-10 вершину 12 хорошей. Наконец, ребро 7-12 проходит через вершину 14 степени $k=5$. Заменим ребро 8-14 ребром 7-12 мы получает вершины 7 и 12 степени $k=5$, продолжая процесс улучшения, мы последовательно заменим 4 ребра.

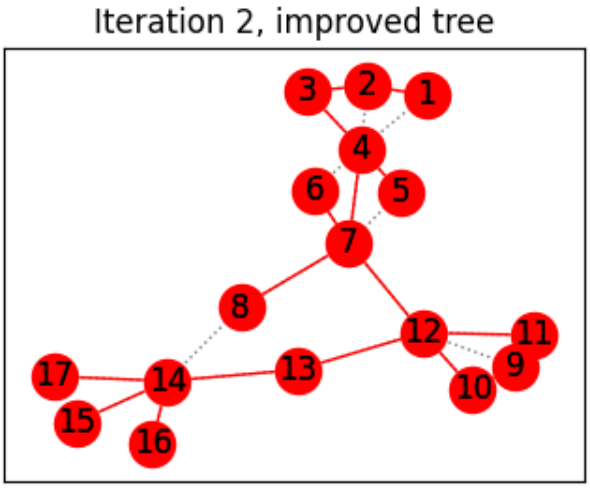
[4]:
```
G_adjdata = ["1 2 4", "2 3 4", "3 4", "4 5 6 7", "5 7", "6 7", "7 8 12",
             "8 14", "9 10 11 12", "10 12", "11 12", "12 13", "13 14",
             "14 15 16 17"]
T_adjdata = ["1 4", "2 4", "3 4", "4 7", "5 7", "6 7", "7 8", "8 14",
             "9 12", "10 12", "11 12", "12 13", "13 14", "14 15 16 17"]
G = nx.parse_adjlist(G_adjdata, nodetype=int)
T = nx.parse_adjlist(T_adjdata, nodetype=int)
np.random.seed(42)
pos = nx.drawing.spring_layout(G)
T, k, W = near_minimum_degree_spanning_tree(
    G, T=T, seed=42, pos=pos, trace=True,
    plot_initial=True, plot_all=True)
```



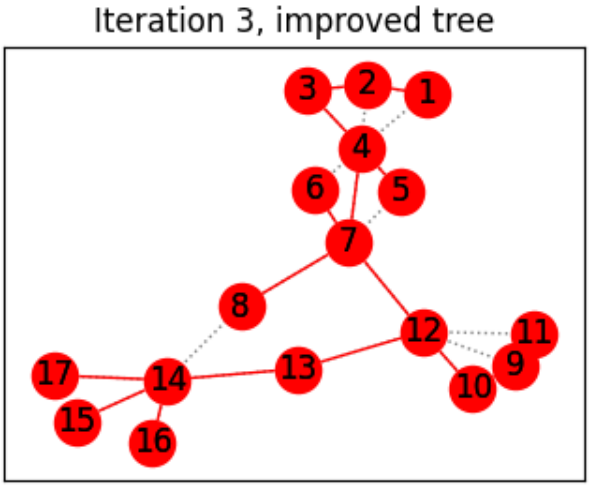

```
Iteration: 1 k: 5, W: {4, 12, 14, 7}
  replace the edge 8-14 with an edge 7-12
  replace the edge 7-5 with an edge 4-5
  replace the edge 9-12 with an edge 9-10
  replace the edge 1-4 with an edge 1-2
```




```
Iteration: 2 k: 4, W: {4, 12, 14, 7}
  replace the edge 2-4 with an edge 2-3
```




```
Iteration: 3 k: 4, W: {4, 12, 14, 7}
  replace the edge 12-11 with an edge 9-11
```



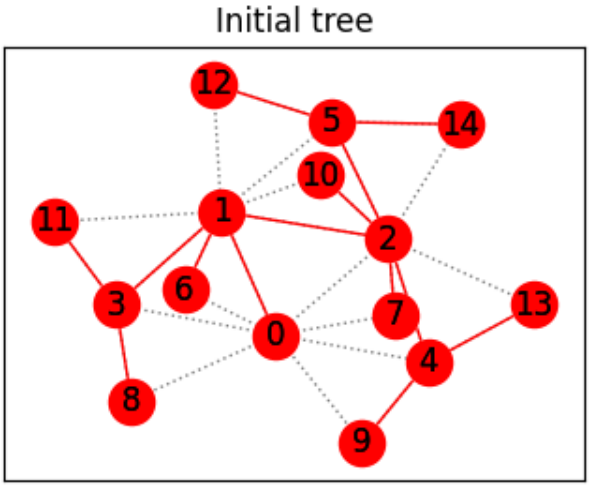

```
Iteration: 4 k: 4, W: {4, 12, 14, 7}
Algorithm finished with k: 4, W: {4, 12, 14, 7}
```

3.6.4 Граф Дороговцева, Гольцева и Мендеса

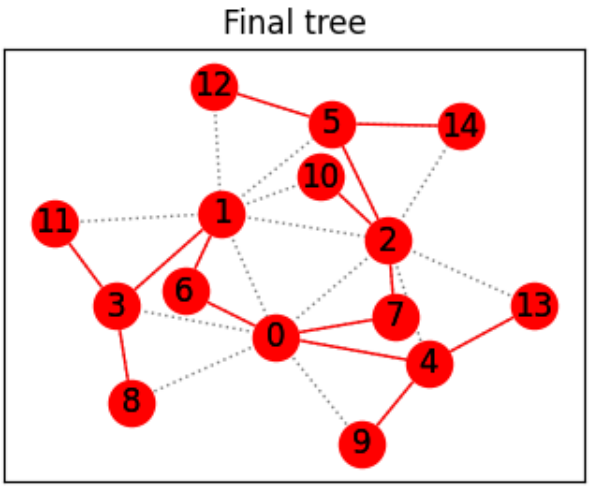
Это иерхаически конструируемый граф, когда на каждом ребре строится треугольник. Он интересен тем, что при больших степенях имеет большую степень для "остовое дерево минимальной степени".

Приведём расчёты для $n = 3$, $n = 4$ и $n = 6$.

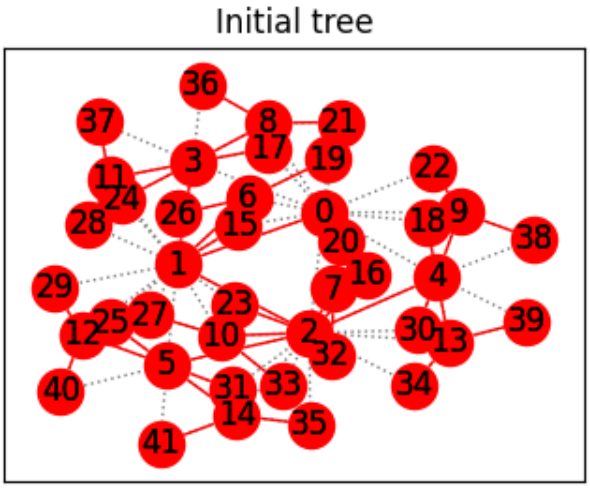
[5]:
```
G = nx.dorogovtsev_goltsev_mendes_graph(3)
T, k, W = near_minimum_degree_spanning_tree(
    G, seed=10, trace=True, plot_initial=True, plot_final=True)
```



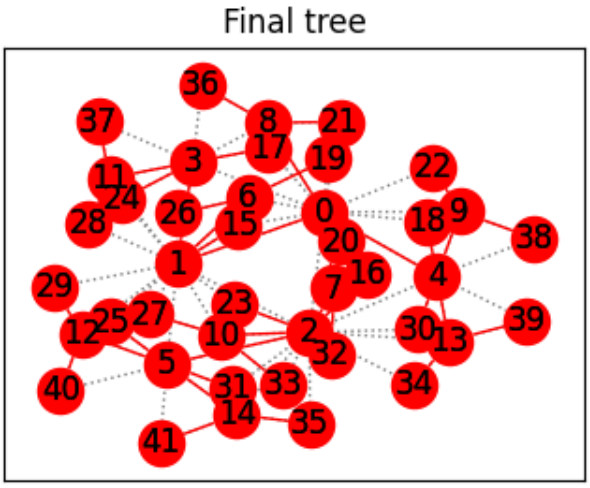
```
Iteration: 1 k: 5, W: {1, 2}
  replace the edge 2-4 with an edge 0-4
Iteration: 2 k: 4, W: {1, 2, 3, 4, 5}
  replace the edge 0-1 with an edge 0-6
Iteration: 3 k: 4, W: {1, 2, 3, 4, 5}
  replace the edge 1-2 with an edge 0-7
Iteration: 4 k: 3, W: {0, 1, 2, 3, 4, 5, 6, 7}
Algorithm finished with k: 3, W: {0, 1, 2, 3, 4, 5, 6, 7}
```



[6]:
```
G = nx.dorogovtsev_goltsev_mendes_graph(4)
T, k, W = near_minimum_degree_spanning_tree(
    G, seed=0, trace=True, plot_initial=True, plot_final=True)
```



```
Iteration: 1 k: 7, W: {2}
  replace the edge 2-4 with an edge 0-4
Iteration: 2 k: 6, W: {1, 2, 3, 4, 5}
  replace the edge 1-2 with an edge 0-7
Iteration: 3 k: 5, W: {1, 2, 3, 4, 5, 7}
  replace the edge 3-8 with an edge 0-8
Iteration: 4 k: 5, W: {0, 1, 2, 3, 4, 5, 7}
Algorithm finished with k: 5, W: {0, 1, 2, 3, 4, 5, 7}
```



[7]:
```
G = nx.dorogovtsev_goltsev_mendes_graph(6)
T, k, W = near_minimum_degree_spanning_tree(
    G, seed=10, trace=True)
```

```
Iteration: 1 k: 11, W: {2}
  replace the edge 2-4 with an edge 0-4
Iteration: 2 k: 10, W: {2, 3, 4, 5}
  replace the edge 2-7 with an edge 0-7
Iteration: 3 k: 9, W: {2, 3, 4, 5}
  replace the edge 1-3 with an edge 0-8
Iteration: 4 k: 9, W: {2, 3, 4, 5, 8}
  replace the edge 0-4 with an edge 0-9
Iteration: 5 k: 9, W: {2, 3, 4, 5, 8, 9}
  replace the edge 2-16 with an edge 0-16
Iteration: 6 k: 9, W: {2, 3, 4, 5, 8, 9}
  replace the edge 5-12 with an edge 1-12
Iteration: 7 k: 8, W: {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14}
  replace the edge 8-3 with an edge 0-17
Iteration: 8 k: 8, W: {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14}
```

```
  replace the edge 9-4 with an edge 0-18
Iteration: 9 k: 8, W: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14}
Algorithm finished with k: 8, W: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
14}
```

3.6.5 Итог практической части:

Был реализован алгоритм, который ищет дерево аппроксимирующее минимальное остовное дерево. Приведён примеры и анализ работы алгоритма. Графы для работы получены из открытой библиотеки.

Список литературы

[1] Garey, M.R.; Johnson, D.S., *Computers and Intractability: A Guide to the Theory of NP-Completeness*, New York: W.H. Freeman, 1979.

[2] Гэри М., Джонсон Д., *Вычислительные машины и труднорешаемые задачи*, М.: Мир, 1982.

[3] Chvátal, Václav, *Tough graphs and Hamiltonian circuits*, Discrete Mathematics, 5 (3): 215–228, 1973

[4] Kou, L.; Markowsky, G.; Berman, L. *A fast algorithm for Steiner trees.* Acta Informatica. 15 (2): 141–145. 1981

[5] Win, Sein *On a connection between the existence ofk-trees and the toughness of a graph.* Graphs and Combinatorics 5, 201–205, 1989

[6] Fürer, Martin; Raghavachari, Balaji, *Approximating the minimum degree spanning tree to within one from the optimal degree.* Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms. SODA, 1992

[7] Fürer, Martin; Raghavachari, Balaji, *Approximating the minimum-degree Steiner tree to within one of optimal*, Journal of Algorithms, 17 (3): 409–423, 1994

[8] Кормен Томас, Лейзерсон Чарльз, Ривест Рональд, Штайн Клиффорд, *Алгоритмы. Построение и анализ*, 3-е издание. : Пер. с англ. — М.: Издательский дом “Вильямс”, 2016.