Министерство образования и науки Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего профессионального образования

**«Уральский федеральный университет**

**имени первого Президента России Б. Н. Ельцина»**

**Институт естественных наук и математики**

**Порождение всех реализаций графического разбиения с точностью до изоморфизма**

Курсовая работа

студента группы КБ-301

Махаева С.В.

Научный руководитель:

профессор кафедры алгебры и

дискретной математики

ИЕНиМ УрФУ,

д.ф.-м.н.

Баранский В.А.

Екатеринбург

2017

**ОГЛАВЛЕНИЕ**

[**Введение** 2](#_Toc481114888)

[**1.** **Описание программы** 3](#_Toc481114889)

[**2.** **Порождение некоторой одной реализации разбиения** 4](#_Toc481114890)

[**3.** **Операция своп** 6](#_Toc481114891)

[**4.** **Изоморфизм разбиений** 7](#_Toc481114892)

[**Заключение** 8](#_Toc481114893)

[**Список литературы** 9](#_Toc481114894)

# **Введение**

Одним из активно развивающихся направлений в современной теории графов является изучение графических разбиений. Для начала дадим определение графического разбиения. Рассмотрим обыкновенный граф *G=(V,E)*. Под графическим разбиением понимают последовательность *(d1,d2,…,dn)*, где *dk∈ℕ* - степень *k*-ой вершины графа *G(k=1,2,…,n)*. Заметим, что такая последовательность задаёт множество графов, ей удовлетворяющих, то есть графов, в которых степень вершины *vi∈V* равна *di (i=1,2,…,n)* для некоторой нумерации вершин из *V*. Такие графы мы будем называть реализациями данного графического разбиения. А разбиения, у которых есть хотя бы одна реализация, будем называть графическими. Наша задача заключается в том, чтобы по данному графическому разбиению построить все её попарно неизоморфные реализации.

Для решения данной задачи в рамках работы была создана программа, которая по заданному графическому разбиению строит все её реализации с точностью до изоморфизма. Описание этой программы содержится в первом параграфе.

Кроме этого, работа включает в себя три параграфа, описывающих последовательно все этапы решения поставленной задачи. Реализованный в рамках исследования алгоритм организует свою работу поэтапно, выделяя подзадачи и разрешая их. В работе эти этапы рассмотрены независимо. Во втором параграфе описано получение одной произвольной реализации данного разбиения. Третий параграф содержит описание операции своп, посредством которой алгоритм порождает все реализации по одной из них. В четвёртом параграфе затронута проблема изоморфизма двух различных реализаций заданного разбиения.

1. **Описание программы**

Программа написана для интерпретатора python 3.5 и находится в открытом доступе по ссылке: <https://github.com/SemyonMakhaev/graphical-sequences>.

В реализации использованы библиотеки *networkx* для структуры графа и *matplotlib.pyplot* для рисования графов, а также для построения графиков.

Использование: *python3 realizations.py [-h] [-dir directory] [-d] degree [degree …]*. Программа создаёт директорию, куда помещает изображения построенных графов в формате *png*.

Позиционные аргументы:

* *degree* – Степень очередной вершины графического разбиения.

Опциональные аргументы:

* *-h, --help* – Вывод справки по использованию.
* *-dir directory, --directory directory* – Директория для вывода картинок.
* *-d, --debug* – Запуск дополнительных проверок.

Реализация основного алгоритма содержится в файле *realizations.py*.

Также для данного решения написаны тесты, они находятся в файле *tests.py*. Запуск тестирующей утилиты: *python3 tests.py*.

Кроме того, для измерения эффективности алгоритма был написан модуль *measure.py*. Он замеряет время работы программы для входных данных заданной величины и строит график, который помещается в директорию *measures*. Запуск: *python3 measures.py [-h] [-s] [length]*.

Позиционные аргументы:

* *length* – Максимальная длина графического разбиения. По умолчанию – *10*. Утилита измерит время работы алгоритма на последовательностях длин от *1* до *length*.

Опциональные аргументы:

* *-h, --help* – Вывод справки по использованию.
* *-s, --save* – Сохранять ли график в файл. Если аргумент не передан, график будет построен и выведен в отдельном окне, где его можно будет сохранить или закрыть.

1. **Порождение некоторой одной реализации разбиения**

Итак, предположим, что нам дано некоторое графическое разбиение. Рассмотрим алгоритм построения одной из реализаций этого разбиения. Сперва в граф добавим столько вершин, сколько элементов в последовательности. Для каждого элемента запоминаем его позицию в разбиении.

Цикл по *i* от *1* до длины разбиения. Если первый его элемент ≤ *0*, то выходим из цикла. Добавляем ребро между вершинами *0* и *i*, вычитаем по *1* из степеней с номерами *0* и *i*. На каждом шаге осуществляем сортировку последовательности по убыванию элементов (степеней). В итоге получаем реализацию исходного графического разбиения.

Отметим также, что если по завершении основного цикла хотя бы один элемент последовательности не равен нулю, то разбиение не является графическим [2].

Приведём листинг метода *get\_realization* из исходного кода программы, который принимает на вход графическое разбиение, строит одну произвольную его реализацию, используя вышеописанный алгоритм, и возвращает полученный граф.

def get\_realization(sequence):

"""Порождает одну реализацию данного разбиения."""

numbered\_sequence = list(sequence)

graph = networkx.Graph()

for i in range(len(sequence)):

numbered\_sequence[i] = [sequence[i], i]

if sequence[i] == 0:

graph.add\_node(i)

for \_ in range(len(numbered\_sequence)):

for idx in range(1, len(numbered\_sequence)):

if numbered\_sequence[0][0] <= 0:

break

if numbered\_sequence[idx][0] > 0 and not graph.has\_edge( \

numbered\_sequence[idx][1], numbered\_sequence[0][1]):

numbered\_sequence[0][0] -= 1

numbered\_sequence[idx][0] -= 1

graph.add\_edge(numbered\_sequence[0][1], numbered\_sequence[idx][1])

numbered\_sequence.sort(key=lambda items: items[0], reverse=True)

for items in numbered\_sequence:

if items[0] != 0:

error('Разбиние не графично')

sys.exit(0)

return graph

1. **Операция своп**

Итак, мы получили одну произвольную реализацию заданного графического разбиения. Она нам ещё потребуется, обозначим её *G*. Теперь рассмотрим алгоритм, с помощью которого мы сможем породить все остальные реализации.

Ключом ко всем остальным графам, удовлетворяющим данному разбиению, служит операция своп (или операция переключения рёбер). Возьмём два несмежных ребра *e1=(a, b)* и *e2=(c, d)* из множества рёбер графа *G* таких, что *(a, c)* и *(b, d)* не являются рёбрами графа. Операция своп преобразует граф *G* к *G′=G-e1-e2+{a, c}+{b, d}* [1]*.* Как видно, операция достаточно бесхитростна: из графа удалили рёбра *e1* и *e2* и добавили в него рёбра *{a, c}* и *{b, d}.* Нетрудно понять, что граф *G′*,полученный в результате применения операции переключения рёбер к *G*, будет реализацией того же графического разбиения, что и *G*, поскольку степени вершин не изменились [1].

Известно, что последовательное применение операций своп к некоторой реализации графического разбиения позволит нам получить все реализации данного разбиения с точностью до изоморфизма [1].

Именно так и устроен метод *get\_all\_realizations*, принимающий на вход граф (реализацию, полученную в предыдущей фазе общего алгоритма), и возвращающий набор всех остальных реализаций: метод последовательно применяет операцию своп для всевозможных подходящих пар несмежных рёбер, начиная с исходного графа. Приведём листинг.

def get\_all\_realizations(realization):

"""Получает все реализации по одной из них."""

stack = [realization]

realizations = [realization]

while len(stack) > 0:

current = stack.pop()

for this\_edge in current.edges():

for that\_edge in current.edges():

if not intersect\_edges(this\_edge, that\_edge):

swapped = swap(current, this\_edge, that\_edge)

if not some(realizations, lambda item: \

networkx.is\_isomorphic(swapped, item)):

stack.append(swapped)

realizations.append(swapped)

return realizations

В приведённом фрагменте кода можно заметить проверку:

networkx.is\_isomorphic(swapped, item)

Действительно, данный метод на каждом шаге выполняет дополнительную проверку на то, не изоморфна ли очередная полученная реализация какой-либо из ранее построенных. Этот нюанс будет рассмотрен в следующем параграфе более подробно.

1. **Изоморфизм разбиений**

Проверка на изоморфизм – один из краеугольных камней современной теории графов. На сегодняшний день науке не известен полиномиальный алгоритм решения задачи проверки на изоморфизм. Стоит, однако, отметить, что предложено много различных алгоритмов решения данной задачи, в том числе, и полиномиальных, но ни для одного из них не доказано, что он решает задачу проверки на изоморфизм точно и в общем случае.

Напомним, что наша задача заключается в том, чтобы породить все реализации графического разбиения *с точностью до изоморфизма*. Таким образом, на каждом шаге основного алгоритма, получив очередную реализацию посредством операции своп, нам необходимо проверить, не построили ли мы раньше реализацию, изоморфную данной. Тем самым мы избавимся от заведомо ненужных нам результатов.

Однако следует заметить, что проверка на изоморфизм – очень сложная операция. Применение её на каждом шаге существенно замедляет работу программы.

В программе используется готовая реализация проверки на изоморфизм *networkx.is\_isomorfic*, определяющая, являются ли изоморфными два переданных графа. Внутри себя данная функция реализует алгоритм VF2, который определяет изоморфизм через всевозможные подграфы и имеет вычислительную сложность в лучшем случае *O(n2)*, а в худшем – *O(n\*n!)* [3]. Разумеется, такая алгоритмическая сложность приводит к очень большому росту времени работы программы на длинных графических разбиениях. Для текущей версии программы были проведены измерения, которые зафиксировали значительный рост времени работы уже для последовательностей длины 11 и выше. К примеру, для разбиений длины 10 и меньше программа выдаёт ответ почти мгновенно, тогда как для последовательности из 15 степеней вершин время ожидания ответа может превышать 4 минуты. Соответствующие графики приведены в директории *measures*.

**Заключение**

Созданная в рамках данной работы программа позволяет достаточно быстро построить все неизоморфные реализации графического разбиения длины не выше 10. Основное время программа проверяет порождённые реализации на изоморфизм.

**Список литературы**

1. Peter L. Erdos, Zoltan Kiraly, Istvan Miklos – «On the swap-distances of different realizations of a graphical degree sequence».
2. Емеличев В.А., Мельников О.И., Сарванов В.И., Тышкевич Р.И. – «Лекции по теории графов».
3. Алгоритм VF2 [Электронный ресурс] – Режим доступа: <https://networkx.github.io/documentation/networkx-1.10/reference/algorithms.isomorphism.vf2.html> (дата обращения: 28.04.2017).