

Правительство Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего профессионального образования
«Национальный исследовательский университет»
«Высшая школа экономики»
Нижегородский кампус

Факультет математики, информатики и компьютерных наук
Кафедра фундаментальной математики

КУРСОВАЯ РАБОТА
РЕАЛИЗАЦИЯ И СРАВНИТЕЛЬНЫЙ АНАЛИЗ
ОПТИМИЗИРОВАННЫХ АЛГОРИТМОВ КРАСКАЛЛА И
ПРИМА

Выполнил:

Студент 2 курса группы 22 ФМ
Турсунов Данил Вячеславович

Научный руководитель:

Малышев Дмитрий Сергеевич

Нижний Новгород
Май 2024 г.

Содержание

1 Введение	4
1.1 Какая задача рассматривается	4
1.2 Какие алгоритмы используются	4
2 Основная часть	5
2.1 Описание	5
2.2 Алгоритм Краскалла	5
2.3 Алгоритм Прима	5
2.4 Оптимизация алгоритма Краскалла	5
2.5 Оптимизация алгоритма Прима	5
2.6 Пример работы алгоритма Прима	5
2.7 Пример работы алгоритма Краскалла	5
3 Вычислительный эксперимент	5
3.1 Генерация данных	5
3.2 Сравнительный анализ	5
4 Заключение	5
4.1 Рассмотренные задачи	5
4.2 Достигнутые результаты	5
5 Алгоритмическая часть	5
5.1 Структура программы	5
5.2 Проверка введённого трёхцветного графа на корректность	5
5.3 Проверка поверхности на ориентируемость	7
5.4 Генератор графов по заданной характеристике Эйлера и числу сёдел	7
5.5 Поиск соседних неподвижных точек	8
5.6 Нахождение сепаратрис	10
5.7 Unit-тестирование	11
6 Графическая часть	12
6.1 Работа с библиотекой Manim	12
6.2 Запуск и результат работы программы	12
7 Ссылка на репозиторий в Github	13
8 Код графической части	13
9 Код алгоритмической части	15
9.1 Поиск в ширину	15
9.2 Проверка на трёхцветность, неориентируемость и отсутствие петель	15
9.3 Проверка на связность	16

9.4	Поиск цикла	16
9.5	Поиск циклов	17
9.6	Проверка на допустимость	18
9.7	Подсчёт Эйлеровой характеристики	18
9.8	Проверка поверхности на ориентируемость	18
9.9	Генератор трёхцветных графов	19
9.10	Поиск соседей	22
9.11	Проверка сепаратрис на пересечение	23
9.12	Поиск сепаратрис 1	24
9.13	Поиск сепаратрис 2	27
10	Список литературы	32

1 Введение

1.1 Какая задача рассматривается

2 Основная часть

2.1 Описание

2.2 Алгоритм Краскалла

2.3 Алгоритм Прима

2.4 Оптимизация алгоритма Краскалла

2.5 Оптимизация алгоритма Прима

2.6 Пример работы алгоритма Прима

2.7 Пример работы алгоритма Краскалла

3 Вычислительный эксперимент

3.1 Генерация данных

3.2 Сравнительный анализ

3.3 Ссылка на Github

4 Заключение

4.1 Рассмотренные задачи

4.2 Достигнутые результаты

5 Список литературы

6 Алгоритмическая часть

6.1 Структура программы

Программа состоит из 2 частей: алгоритмической и графической, каждая из частей запускается отдельно и независимо друг от друга. Изначально запускается алгоритмическая часть на языке C++, туда вводится корректный трёхцветный граф, программа его обрабатывает и выдаёт в отдельном файле координаты сепаратрис. Далее запускается графическая часть программы, написанная на языке Python. Она считывает координаты из файла и генерирует по

заданным координатам конца и начала сепаратрис 3D-изображение, а затем, после рендеринга в библиотеке Manim, показывает её пользователю.

6.2 Проверка введённого трёхцветного графа на корректность

Проверяет граф на корректность функция *is_acceptable*, которая принимает на вход заданный граф и возвращает булево значение: True, если граф является корректным (допустимым) трёхцветным графом, и False, если граф таковым не является.

Согласно определению 10 граф называется корректым трёхцветным графом, если:

- 1) граф является трёхцветным, то есть попадает под определение трёхцветности;
- 2) граф является связным;
- 3) все SU-циклы в графе имеют длину равную четырём.

Функция для проверки пункта 1 вызывает функцию *is_3_colored_and_non_oriented*, которая действует следующим образом: функция циклом проходит по вершинам графа, для каждой вершины проверяет, действительно ли из неё выходит только 3 ребра, причем эти рёбра должны быть разных цветов: *u*, *s* и *t*. Параллельно с этим в этом же цикле проверяется то, что граф является неориентируемым, а также то, что граф не содержит петель, то есть циклов длины 1. Если хотя бы одно из условий не выполняется, функция возвращает False, в противном случае она возвращает True.

Для проверки пункта 2 функция вызывает функцию *is_connected*, которая считает расстояния от вершины с порядковым номером 0 до остальных вершин при помощи функции *bfs*. Здесь под расстоянием имеется в виду длина кратчайшего пути между 2 вершинами. Если не существует пути, соединяющего 2 вершины, расстояние считается равным *inf*. Если хотя бы одно расстояние окажется равным *inf*, то граф не связан и функция вернёт значение False, а в противном случае граф является связным и функция возвращает значение True.

Функция *bfs* принимает на вход начальную вершину (в данном случае это вершина с номером 0) и сам граф и работает по алгоритму *breadth – first search*, что можно перевести как «поиск в ширину». Алгоритм работает с применением структуры данных «очередь». Принцип работы этой структуры данных объясняется фразой: «Первый зашёл - последний вышел.». Алгоритм работает следующим образом: создаётся вектор, структура данных для хранения данных в C++, содержащий известные расстояния от начальной вершины до остальных, изначально заполнен *inf*, кроме начальной, так как расстояние от начальной до начальной вершины равно 0, и очередь, состоящая только из начальной вершины, далее запускается цикл, он работает до тех пор, пока очередь не опустеет. В каждой итерации цикл делает первую вершину из очереди текущей и проходит по всем соседним вершинам текущей (то есть тем, кто соединён с ней ребром), и, если известное расстояние до соседа больше суммы расстояния до текущей и единицы (она появляется за счёт ребра, которое их соединяет), то минимальное известное расстояние обновляется, а вершина-сосед кладётся в очередь. Функция возвращает вектор расстояний до каждой из вершин.

Для проверки пункта 3 функция вызывает функцию *find_cycles* с переданными в неё графом, литералами «s» и «u», которые отвечают за то, какого цвета циклы надо найти. Функция

find_cycles возвращает вектор, состоящий из циклов, каждый цикл представляет собой последовательность номеров вершин цикла, так же последовательно соединённых между собой в самом цикле. Эта функция в процессе своего исполнения использует факт, который вытекает из определений 7 и 8 про то, что каждая отдельно взятая вершина лежит только в одном *su*–, одном *tu*– и одном *st*–цикле, поэтому каждый цикл по отдельности ищется функцией *find_cycle* достаточно тривиальным алгоритмом, который просто идёт по циклу из начальной вершины, пока снова не встретит начальную вершину. Далее функция *is_acceptable* проверяет, имеют ли все SU-циклы в графе длину 4.

Если все условия выполнены, функция *is_acceptable* возвращает True, в противном случае возвращает False.

6.3 Проверка поверхности на ориентируемость

Для построения алгоритма, проверяющего поверхность, на которой задан градиентно-подобный каскад, по трёхцветному графу, потребуется пункт 2 теоремы 2, который гласит о том, что поверхности ориентируема тогда и только тогда, когда все циклы графа имеют чётную длину.

Сделать вывод о четной длине всех циклов графа можно найдя хотя бы один цикл нечётной длины. Для нахождения такого цикла потребуется небольшой экскурс в теорию, связанную с базой циклов и алгоритмом её нахождения.

Базой циклов неориентированного графа является такой набор циклов, путём соединения или вычитания которых могут получиться все остальные циклы. Для нахождения базы циклов необходимо построить из графа так называемое «переплетающееся дерево» (spanning tree), то есть просто выбрать какую-нибудь вершину за корень дерева, а потом идти по нему уже упомянутым выше алгоритмом поиска в ширину из корневой вершины, при этом вместо поиска расстояний отмечать посещённые вершины, если вершина-сосед не посещена, то ребро, связывающее текущую вершину с ней, добавлять в «переплетающееся дерево», а далее, путём добавления по одному в дерево рёбер изначального графа, которые в дерево не попали, по одному найти все циклы. Эти циклы и будут составлять базу циклов в графе. Очевидно, что при вычитании или сложении циклов чётной длины получится цикл чётной длины, то есть на чётность достаточно проверить всего лишь циклы из базы циклов.

Алгоритм, находящий «переплетающееся дерево» и сразу проверяющий циклы базы циклов на чётную длину, реализован в функции *is_oriented_surface*, которая возвращает True, если поверхность ориентируема, и False, если поверхность неориентируема.

6.4 Генератор графов по заданной характеристике Эйлера и числу сёдел

Для дальнейшей проверки алгоритма на корректность потребуется генерировать корректные трёхцветные графы со всевозможным количеством стоков и источников, такие, что соответствующие им градиентно-подобные каскады расположены на сфере, по заданному числу сёдел σ . Изложенную ниже генерацию можно обобщить для генерации трёхцветных графов,

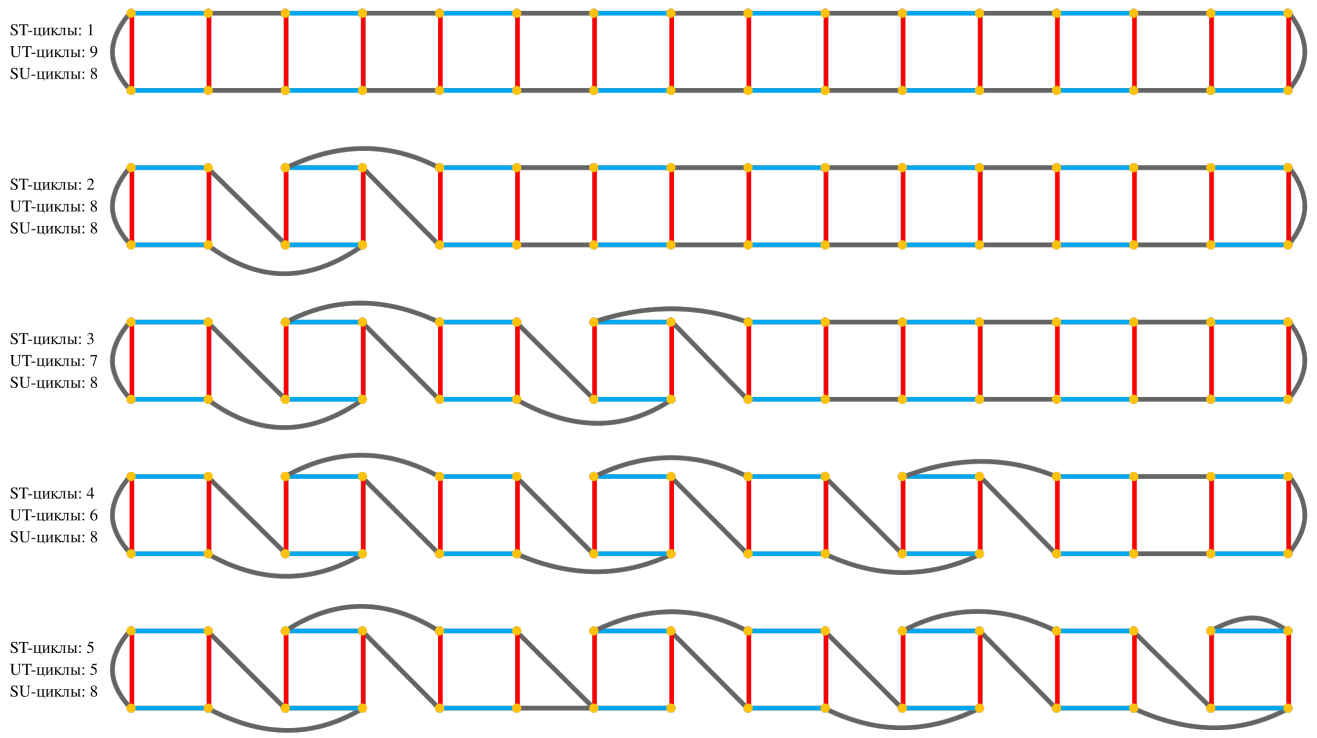


Рис. 1: Генератор графов для $e = 2$ и $\sigma = 8$

такие, что соответствующие им градиентно-подобные каскады расположены на ориентируемой поверхности, определяемой характеристикой Эйлера e .

Из определения 10 известно, что длина SU-циклов корректного трёхцветного графа равна 4, причём количество SU-циклов равно числу сёдел σ . Тогда расположим «квадраты» SU-циклов, верхнее и нижнее ребро которых имеют одинаковый цвет для всех «квадратов» (будем считать, что верхние и нижние ребра - s-рёбра), в ряд и будем проводить из каждой вершины рёбра цвета t . Далее будем соединять t -ребром с ближайшей вершиной ближайшего соседнего «квадрата» SU-цикла, кроме первых 2 и последних 2 вершин, которые соединяются между собой соответственно. Получили тривиальный пример графа с числом SU-циклов, равному σ , числом ST-циклов, равному 1, и числом UT-циклов, равному $\sigma + 1$.

Увеличим количество ST-циклов на 1 и уменьшим количество UT-циклов на 1. Это можно сделать из построенного выше тривиального графа при помощи переподвязок t -циклов. Переподвязывать t -циклы будем следующим способом: возьмём чётный по счёту «квадрат» и перекрасим s-рёбра в u-рёбра и наоборот. Одна такая переподвязка увеличивает количество ST-циклов на 1 и уменьшает количество UT-циклов на 1.

Проводя такие переподвязки на чётных «квадратах» по одной поверх друг друга, сгенерируем циклы, для которых количество ST-циклов принимает значения (с учётом тривиального графа) $\{1, \dots, [\sigma/2]\}$, а количество UT-циклов $\{[(\sigma + 1)/2], \dots, \sigma + 1\}$.

Ниже приведены примеры построения трёхцветных графов при $\sigma = 8$ и $e = 0$ и $e = -2$ для рис. 3 и для рис. 4 соответственно.

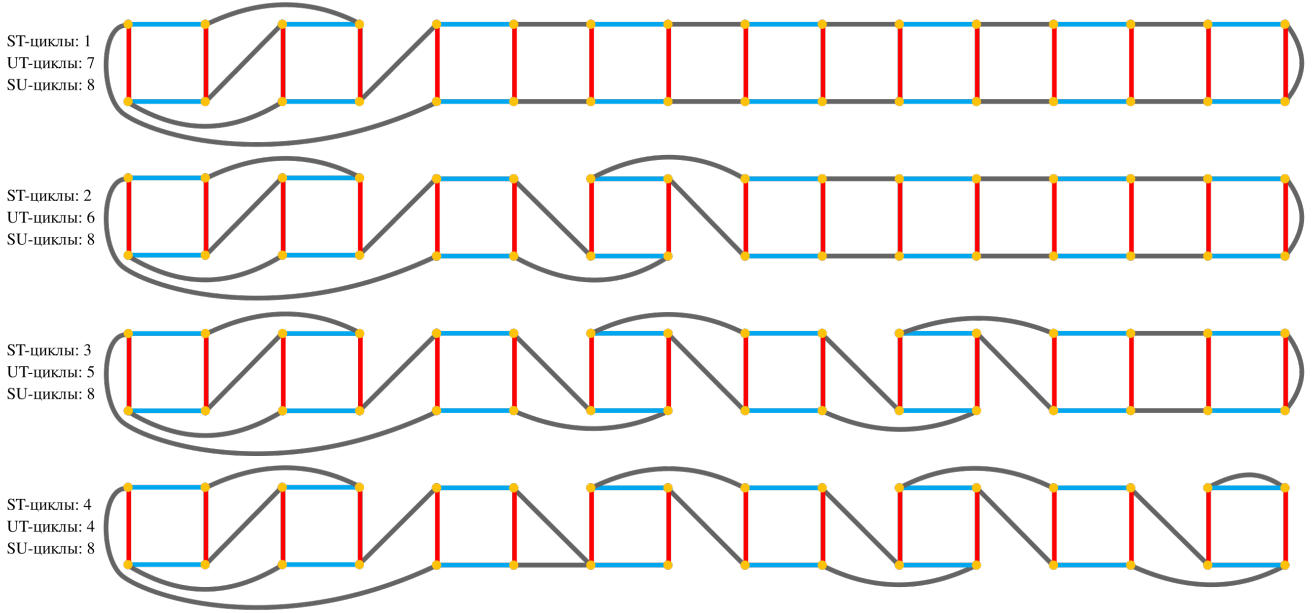


Рис. 2: Генератор графов для $e = 0$ и $\sigma = 8$

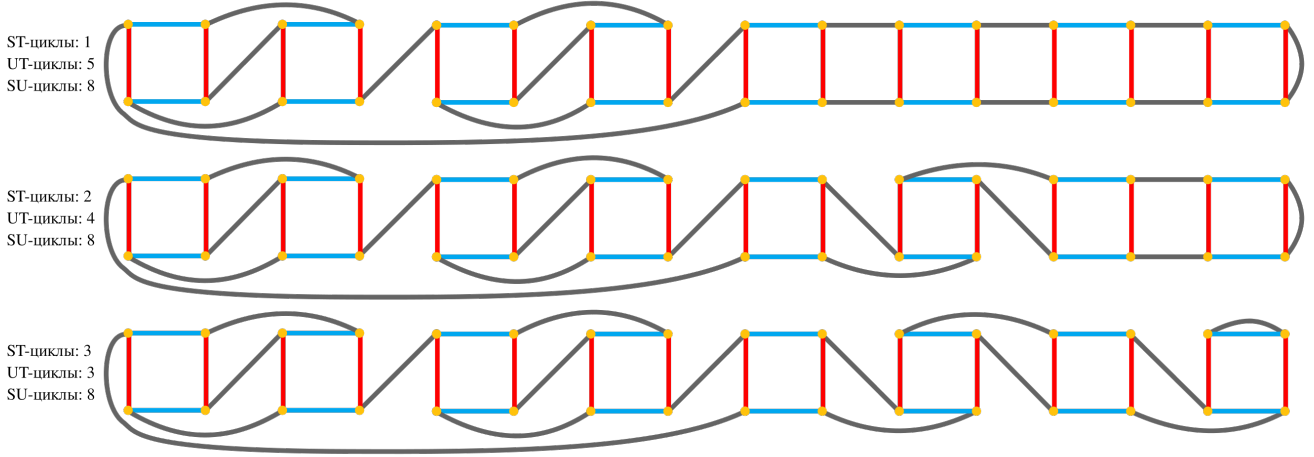


Рис. 3: Генератор графов для $e = -2$ и $\sigma = 8$

6.5 Поиск соседних неподвижных точек

Нам дан корректный трёхцветный граф, для дальнейшей визуализации динамической системы для каждой неподвижной точки необходимо найти соседние, то есть соединённые с данной неподвижной точкой сепаратрисой, неподвижные точки. Для этого реализуем функцию *find_neighbors*, которая принимает на вход корректный трёхцветный граф, а выдаёт вектор, состоящий из стоков, сёдел и источников, а также их соседей в правильной последовательности.

Для начала найдём все ST-, UT-, SU-циклы в исходном графе. Напомним, что каждый ST-цикл соответствует источнику, UT-цикл - стоку, SU-цикл - седлу. Из построения трёхцветного графа следует, что циклы имеют общее красное или синее ребро тогда и только тогда, когда неподвижные точки, представляющие эти циклы, соединены сепаратрисой, причём порядок обхода сепаратрис вокруг неподвижной точки соответствует порядку обхода рёбер графа, а также что одно ребро лежит ровно в двух двухцветных циклах, поэтому, найдя все двухцветные цик-

лы, будем идти по ним в порядке обхода и для красных и синих рёбер будем смотреть, в каком ещё двухцветном цикле они лежат, далее сопоставляем новому двухцветному циклу для ребра неподвижную точку, соответствующую этому циклу. Прделаем это для всех неподвижных точек, получим искомый вектор.

6.6 Нахождение сепаратрис

Представим сферу как прямоугольник $[-90, 90] \times [0, 360]$, где все точки из отрезка -90 х $[0, 360]$ и из отрезка 90 х $[0, 360]$ отождествлены между собой. Впоследствии при визуализации этот прямоугольник будем отображать на сферу по формуле:

$$\begin{cases} x = r * \sin(\psi) * \cos(\phi) \\ x = r * \sin(\psi) * \sin(\phi) \\ x = r * \cos(\psi) \end{cases}$$

Сепаратрисы будем представлять как пары, состоящие из цвета сепаратрисы, красная или синяя, и вектора координат, содержащего $a, a0, b, b0$ и при этом заданной формулой:

$$\begin{cases} x = a * t + a_0 \\ y = b * t + b_0 \end{cases}$$

где $t \in [0, 1]$.

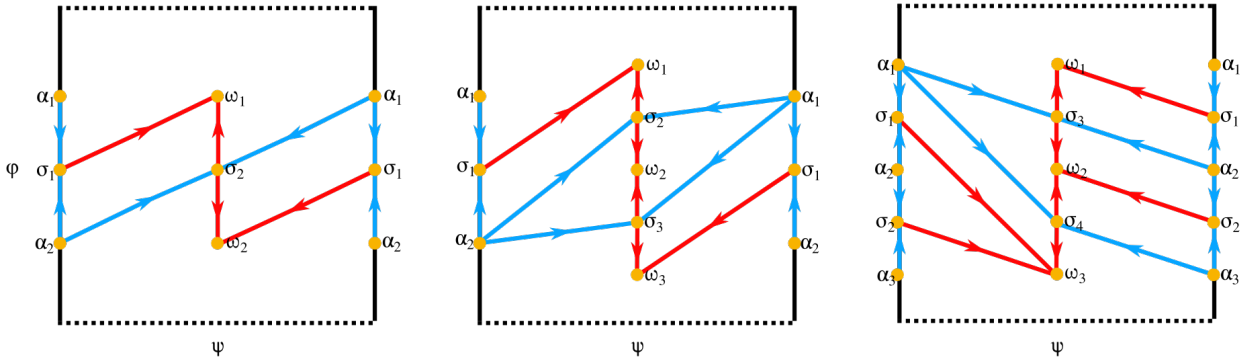


Рис. 4: Представление каскада на сфере.

Для нахождения координат сепаратрис на сфере реализована функция *find_separatres_coords*, она принимает на вход изначальный граф. Функция сначала запускает описанную выше функцию *find_neighbors* и находит соседние неподвижные точки для каждой неподвижной точки. Напомним, что возвращаемый функцией *find_neighbors* вектор устроен так, что он для каждой неподвижной точки содержит информацию о том, чем является неподвижная точка: стоком, источником или седлом. Функция *find_separatres_coords* проходит по вектору с соседями и выявляет набор источников и сёдел, которые должны быть расположены на краях прямоугольника, расположенных последовательно. Делает она это так:

заходим в первый источник, ищем соседа-седло, далее для этого седла ищем соседа-источник, которого у нас ещё нет в последовательности, далее для источника ищем соседа-седло, которого ещё не посетили, так повторяем до тех пор, пока не пройдем через все источники. После этого, функция аналогично выявляет расположенные последовательно оставшиеся стоки и седла, которые должны лежать в центре прямоугольника. Параллельно с этим функция для каждой неподвижной точки запоминает её координаты на этом прямоугольнике.

После того, как координаты неподвижных точек найдены, функция начинает поиск координат сепаратрис. Для этого используется вспомогательная функция *find_sep*. Она сначала находит координаты сепаратрис, расположенных по центру и с краю, там достаточно тривиальное расположение сепаратрис. После этого, функция пытается нарисовать сепаратрисы слева от центра: если отдельно взятая сепаратриса не пересеклась с остальными сепаратрисами, нарисованными справа, то рисуем её, если пересеклась, то переносим её в правую от центра часть. Для правой части отрисовываем все сепаратрисы, если какая-то пересеклась, то отражаем порядок и координаты неподвижных точек, расположенных по центру, и отрисовываем заново.

Полученные координаты сепаратрис уже отдельно от файла, в файле *main.cpp*, передаём в текстовый документ, который затем будет прочтён из файла с визуализацией.

Пересечение сепаратрис находится при помощи функции *is_separatres_cross*, которая принимает на вход координаты двух сепаратрис, которые необходимо проверить на пересечение. Проверка сепаратрис на пересечение производится при помощи перебора всех возможных случаев (таких как пересечение, совпадение, параллельность) и переходом от параметрического задания прямой к каноническому, что при приравнивании поможет найти точку пересечения этих прямых. Если точка лежит на отрезке, то сепаратрисы пересеклись, если нет, то не пересеклись. При пересечении функция возвращает булево значение *True*, в противном случае она возвращает *False*.

6.7 Unit-тестирование

Для стабильной работы программы при изменении в дальнейшем в каталоге под названием *graph_algorithms_unit_test* содержатся файлы, в которых написаны unit-тесты для различных функций. Если при изменении программа будет работать неправильно, то unit-тесты распознают аномалии.

7 Графическая часть

7.1 Работа с библиотекой Manim

Графическая часть программы, написанная в файле `draw.py`, отвечает за генерацию 3D-изображения дискретной динамической системы на сфере.

В файле определён класс `DynamicalSystemSphere`, который в дальнейшем будет указываться при запуске графической части.

Графическая часть работает по следующему алгоритму:

1) Считывается информация о сепаратрисах, полученная в результате запуска алгоритмической части;

2) Объявляется функция `func_sphere`, задающая параметрически поверхность сферы:

$$\begin{cases} x = r * \cos(u) * \sin(v) - x_0 \\ y = r * \sin(u) * \sin(v) - y_0 \\ z = r * \cos(v) - z_0 \end{cases}$$

3) Объявляется функция `construct`, которая отвечает непосредственно за генерацию 3D-изображения. При помощи библиотеки `Manim` создаются поверхность сферы и оси `OX`, `OY` и `OZ`. Далее каждая сепаратриса, разбитая на 3 равные части, отличающиеся по цвету: более яркий красный и синий цвет соответствуют близости к стокам и источникам соответственно, а бледные оттенки этих цветов соответствуют близости к седлу, по отдельности добавляется на поверхность сферы.

Сепаратриса, заданная параметрически на прямоугольнике значениями a, a_0, b, b_0 , отображается на поверхность сферы по правилу:

$$\begin{cases} x = \cos(\pi * (t * b + b_0)/180) * \sin(\pi * (t * a + a_0)/180) \\ y = \cos(\pi * (t * b + b_0)/180) * \cos(\pi * (t * a + a_0)/180) \\ z = \sin(\pi * (t * b + b_0)/180) \end{cases}$$

где t принадлежит $[0, 1]$ (причём для создания более «говорящей» картинки для различных оттенков сепаратрисы отрезок разбивается на 3 равные части, каждая из которых отображается независимо от остальных).

Для получения объёмного и «говорящего» изображения объявляется полный оборот камеры вокруг сферы.

7.2 Запуск и результат работы программы

Для запуска генерации 3D-изображения необходимо:

1) При помощи терминала установить библиотеку `Manim` на компьютер, если эта библиотека ещё не установлена.

2) Предварительно запустить алгоритмическую часть с введённым в неё корректным трёхцветным графом.

3) При помощи терминала перейти в каталог с файлом draw.py.

4) Запустить графическую часть при помощи команды:

manim -pqh draw.py DynamicalSystemSphere - для генерации изображения высокого качества;

manim -pql draw.py DynamicalSystemSphere - для генерации изображения низкого качества.

8 Ссылка на репозиторий в Github

Ссылка на репозиторий: https://github.com/dan1lka257/graphs_and_algorithms/tree/main

9 Код графической части

```
1 from manim import *
2
3 class DynamicalSystemSphere(ThreeDScene):
4     separatress = []
5     with open('../separatres.txt') as file:
6         lines = [line.rstrip() for line in file]
7     for line in lines:
8         separatress.append([line[0]] + list(map(int, line[2:].split(' '))))
9
10    def func_sphere(self, u, v, radius = 1):
11        sphereCentre = np.array([0, 0, 0])
12        return radius * np.array([np.cos(u) * np.sin(v) - sphereCentre[0]/radius,
13            np.sin(u) * np.sin(v) - sphereCentre[1]/radius,
14            np.cos(v) - sphereCentre[2]/radius])
15
16    def construct(self):
17        rng = 7
18        axes = ThreeDAxes(
19            x_range=[-rng, rng], y_range=[-rng, rng], z_range=[-rng, rng],
20            x_length=rng, y_length=rng, z_length=rng
21        )
22        x_label = axes.get_x_axis_label(Tex("x"))
23        y_label = axes.get_y_axis_label(Tex("y"))
24        z_label = axes.get_z_axis_label(Tex("z"))
25        mainSphere = Surface(
26            lambda u, v: axes.c2p(*self.func_sphere(u, v, rng)),
27            u_range=[0, TAU],
28            v_range=[0, TAU],
29            resolution=32, fill_opacity=0.1, checkerboard_colors=['#29ABCA', '#236B8E'], stroke_color=BLACK, stroke_width=0.1
30        )
```

```

31     for sep in self.separatress:
32         u_sep1 = ParametricFunction(
33             lambda t: (0.5 * rng) * np.array([
34                 np.cos(np.pi * (t * sep[3] + sep[4]) / 180) * np.sin(np.pi * (t * sep[1] +
35                 sep[2]) / 180),
36                 np.cos(np.pi * (t * sep[3] + sep[4]) / 180) * np.cos(np.pi * (t * sep[1] +
37                 sep[2]) / 180),
38                 np.sin(np.pi * (t * sep[3] + sep[4]) / 180)
39             ]), color=RED_A if sep[0]=='u' else BLUE_A, t_range = np.array([0, 1/3,
40             0.01])
41         )
42         u_sep1.set_z_index(mainSphere.z_index)
43         self.add(u_sep1)
44
45         u_sep2 = ParametricFunction(
46             lambda t: (0.5 * rng) * np.array([
47                 np.cos(np.pi * (t * sep[3] + sep[4]) / 180) * np.sin(np.pi * (t * sep[1] +
48                 sep[2]) / 180),
49                 np.cos(np.pi * (t * sep[3] + sep[4]) / 180) * np.cos(np.pi * (t * sep[1] +
50                 sep[2]) / 180),
51                 np.sin(np.pi * (t * sep[3] + sep[4]) / 180)
52             ]), color=RED_C if sep[0]=='u' else BLUE_C, t_range = np.array([1/3, 2/3,
53             0.01])
54         )
55         u_sep2.set_z_index(mainSphere.z_index)
56         self.add(u_sep2)
57
58         u_sep3 = ParametricFunction(
59             lambda t: (0.5 * rng) * np.array([
60                 np.cos(np.pi * (t * sep[3] + sep[4]) / 180) * np.sin(np.pi * (t * sep[1] +
61                 sep[2]) / 180),
62                 np.cos(np.pi * (t * sep[3] + sep[4]) / 180) * np.cos(np.pi * (t * sep[1] +
63                 sep[2]) / 180),
64                 np.sin(np.pi * (t * sep[3] + sep[4]) / 180)
65             ]), color=RED_E if sep[0]=='u' else BLUE_E, t_range = np.array([2/3, 1,
66             0.01])
67         )
68         u_sep3.set_z_index(mainSphere.z_index)
69         self.add(u_sep3)
70
71     # u_sep.rotate(PI / 4, about_point=[0, 0, 0], axis=RIGHT)
72     self.add(axes, mainSphere, x_label, y_label, z_label)
73     self.set_camera_orientation(theta=75*DEGREES, phi=75*DEGREES)
74     self.begin_ambient_camera_rotation(rate=PI/10, about='theta')
75     self.wait(10)
76     self.stop_ambient_camera_rotation()

```

10 Код алгоритмической части

10.1 Поиск в ширину

```
1 vector<int> bfs(int s, vector<vector<pair<int, char>>>& graph) {
2     // BFS which find distance to other vertices
3     int n = graph.size();
4     vector<int> dist(n, INF);
5     dist[s] = 0;
6     queue<int> q;
7     q.push(s);
8
9     while (!q.empty()) {
10        int v = q.front();
11        q.pop();
12        for (pair<int, char> u : graph[v]) {
13            if (dist[u.first] > dist[v] + 1) {
14                dist[u.first] = dist[v] + 1;
15                q.push(u.first);
16            }
17        }
18    }
19
20    return dist;
21 }
```

10.2 Проверка на трёхцветность, неориентируемость и отсутствие петель

```
1 bool is_3_colored_and_non_oriented(vector<vector<pair<int, char>>>& graph) {
2     // checks graph on tricolor and orientedness
3     // 1 edge of each of 3 colors in one verticle
4     for (int i = 0; i < graph.size(); i++) {
5         // tricolor check
6         int color_u = 0, color_s = 0, color_t = 0;
7         for (int j = 0; j < graph[i].size(); j++) {
8             if (graph[i][j].second == 'u') color_u += 1;
9             if (graph[i][j].second == 's') color_s += 1;
10            if (graph[i][j].second == 't') color_t += 1;
11        }
12        // orientedness check
13        int oriented = 1;
14        for (int j = 0; j < graph[i].size(); j++) {
15            if (i != graph[graph[i][j].first][j].first) {
16                oriented = 0;
17            }
18        }
19    }
20 }
```

```

18     }
19     // no cycles with len = 1
20     int has_no_cycles_with_len_1 = 1;
21     for (int j = 0; j < graph[i].size(); j++) {
22         if (i != graph[i][j].first) {
23             has_no_cycles_with_len_1 = 0;
24         }
25     }
26     if (!(color_u == 1 && color_s == 1 && color_t == 1)) {
27         // Graph is not 3 colored
28         return false;
29     }
30     if (!(oriented)) {
31         // Graph is not oriented
32         return false;
33     }
34
35     if (!(has_no_cycles_with_len_1)) {
36         // Graph has cycles with len = 1
37         return false;
38     }
39 }
40 return true;
41 }

```

10.3 Проверка на связность

```

1 bool is_connected(vector<vector<pair<int, char>>>& graph) {
2     // checks graph is connected or not
3     vector<int> dist = bfs(0, graph);
4     int max_dist = 0;
5     for (int j = 0; j < dist.size(); j++) {
6         max_dist = max(max_dist, dist[j]);
7     }
8     if (max_dist == INF) {
9         return false;
10    }
11    return true;
12 }

```

10.4 Поиск цикла

```

1 vector<int> find_cycle(char color1, char color2, char prev_color, int cur_v, int
    prev_v, vector<vector<pair<int, char>>>& graph, vector<string>& color, vector<
    int>& parent) {
2     // Standard cycle finding taking into account cycle's bicolor
3     color[cur_v] = "grey";

```

```

4  for (int i = 0; i < graph[cur_v].size(); i++) {
5      if (graph[cur_v][i].second != prev_color && (graph[cur_v][i].second == color1
        || graph[cur_v][i].second == color2)) {
6          if (color[graph[cur_v][i].first] == "white") {
7              parent.push_back(cur_v);
8              return find_cycle(color1, color2, graph[cur_v][i].second, graph[cur_v][i]
9                  ].first, cur_v, graph, color, parent);
10             }
11             if (color[graph[cur_v][i].first] == "grey" && graph[cur_v][i].first !=
12                 prev_v && cur_v != prev_v) {
13                 parent.push_back(cur_v);
14                 return parent;
15             }
16         }
17     }
18     parent.push_back(cur_v);
19     color[cur_v] = "black";
20     return parent;
21 }

```

10.5 Поиск циклов

```

1  vector<vector<int>> find_cycles(vector<vector<pair<int, char>>>& graph, char
2      color1, char color2) {
3      // Find bicolored cycles in graph
4      int n = graph.size();
5      vector<int> used_v(n, 0);
6      vector<vector<int>> cycles;
7      for (int i = 0; i < n; i++) {
8          if (!used_v[i]) {
9              vector<string> color(n, "white");
10             vector<int> parent;
11             vector<int> cycle = find_cycle(color1, color2, 'o', i, i, graph, color,
12                 parent);
13             if (cycle.empty()) {
14                 continue;
15             }
16             for (int j = 0; j < cycle.size(); j++) {
17                 used_v[cycle[j]] = 1;
18             }
19             cycles.push_back(cycle);
20         }
21     }
22     return cycles;
23 }

```


10.6 Проверка на допустимость

```
1 bool is_acceptable(vector<vector<pair<int, char>>>& graph) {
2     // Length of su-cycle is equal 4, graph is connected, non_oriented and
3     // tricolored
4     vector<vector<int>> cycles = find_cycles(graph, 's', 'u');
5     bool su_cycle_is_4 = 1;
6     for (int i = 0; i < cycles.size(); i++) {
7         if (cycles[i].size() != 4) {
8             su_cycle_is_4 = 0;
9         }
10    }
11    return su_cycle_is_4 && is_connected(graph) && is_3_colored_and_non_oriented(
12        graph);
13 }
```

10.7 Подсчёт Эйлеровой характеристики

```
1 int count_euler_number(vector<vector<pair<int, char>>>& graph) {
2     // Euler number = tu - su + st
3     return find_cycles(graph, 'u', 't').size() - find_cycles(graph, 's', 'u').size()
4         + find_cycles(graph, 's', 't').size();
5 }
```

10.8 Проверка поверхности на ориентируемость

```
1 bool is_oriented_surface(vector<vector<pair<int, char>>>& graph) {
2     // Graph is on oriented surface <=> length of all cycles is even
3     // All cycles is even <=> any cycle from base of cycles is even
4     vector<vector<int>> spanning_tree(graph.size(), vector<int> ());
5     vector<int> visited(graph.size(), 0);
6     // Creating spanning tree for future cycle finding
7     for (int i = 0; i < graph.size(); i++) {
8         vector<int> i_neighbors;
9         for (int j = 0; j < graph[i].size(); j++) {
10            if (visited[graph[i][j].first] == 0) {
11                spanning_tree[i].push_back(graph[i][j].first);
12                spanning_tree[graph[i][j].first].push_back(i);
13                i_neighbors.push_back(j);
14                visited[graph[i][j].first] = 1;
15                visited[i] = 1;
16            }
17        }
18    }
19    vector<int> way_from_root = bfs(0, spanning_tree);
20    // Finding base of cycles & checking if cycles is even
21    vector<vector<int>> base;
```

```

22 bool is_even_base = true;
23 for (int i = 0; i < graph.size(); i++) {
24     for (int j = 0; j < graph[i].size(); j++) {
25         bool is_in = false;
26         for (int k = 0; k < spanning_tree[i].size(); k++) {
27             if (graph[i][j].first == spanning_tree[i][k]) {
28                 is_in = true;
29                 break;
30             }
31         }
32         if (!is_in && (way_from_root[i] + way_from_root[graph[i][j].first] + 1) % 2
33 == 1) {
34             is_even_base = false;
35             cout << "!!!" << i << ":" << way_from_root[i] << " " << graph[i][j].first
36 << ":" << way_from_root[graph[i][j].first] << "!!!";
37             break;
38         }
39     }
40     if (!is_even_base) {
41         break;
42     }
43 }
44 return is_even_base;
45 }

```

10.9 Генератор трёхцветных графов

```

1 vector<vector<vector<pair<int, char>>>> graph_generator(int euler_number, int
  saddles) {
2     vector<vector<vector<pair<int, char>>>> graphs(0);
3     // Generating primal graph
4     vector<vector<pair<int, char>>> graph(0);
5     for (int i = 0; i < saddles * 4; i++) {
6         vector<pair<int, char>> neighbours;
7         if (i == 0) { // first verticle
8             neighbours = { {i + 1, 't'}, {i + 2, 's'}, {i + 1, 'u'} };
9         } else if (i == 1) { // second verticle
10            neighbours = { {i - 1, 't'}, {i + 2, 's'}, {i - 1, 'u'} };
11        } else if (i == saddles * 4 - 1) { // last verticle
12            neighbours = { {i - 1, 't'}, {i - 2, 's'}, {i - 1, 'u'} };
13        } else if (i == saddles * 4 - 2) { // penultimate verticle
14            neighbours = { {i + 1, 't'}, {i - 2, 's'}, {i + 1, 'u'} };
15        } else if (i % 4 == 0) {
16            neighbours = { {i - 2, 't'}, {i + 2, 's'}, {i + 1, 'u'} };
17        } else if (i % 4 == 1) {
18            neighbours = { {i - 2, 't'}, {i + 2, 's'}, {i - 1, 'u'} };
19        } else if (i % 4 == 2) {
20            neighbours = { {i + 2, 't'}, {i - 2, 's'}, {i + 1, 'u'} };

```

```

21     } else if (i % 4 == 3) {
22         neighbours = { {i + 2, 't'}, {i - 2, 's'}, {i - 1, 'u'} };
23     }
24     graph.push_back(neighbours);
25 }
26 // operating with euler number
27 if (2 - euler_number > saddles) {
28     cout << "Uncorrect input: number of saddles must be lower or equal than (2 -
29     euler_number)\n";
30     return graphs;
31 }
32 for (int i = euler_number; i < 2; i += 2) {
33     int j = 8 * (i - euler_number) / 2;
34     if (euler_number == 0 && saddles == 2) { // usual torus
35         graph[j][0].first = j + 1;
36         graph[j + 1][0].first = (j + 1) - 1;
37         graph[j + 2][0].first = (j + 2) + 4;
38         graph[j + 3][0].first = (j + 3) + 1;
39         graph[j + 4][0].first = (j + 4) - 1;
40         graph[j + 5][0].first = (j + 5) + 2;
41         graph[j + 6][0].first = (j + 6) - 4;
42         graph[j + 7][0].first = (j + 7) - 2;
43     } else if (i == euler_number) { // First step
44         if (2 - euler_number < saddles) {
45             graph[j][0].first = 8 * (2 - euler_number) / 2 + 1;
46             graph[8 * (2 - euler_number) / 2 + 1][0].first = j;
47         } else {
48             graph[j][0].first = 8 * (2 - euler_number) / 2 - 1;
49             graph[8 * (2 - euler_number) / 2 - 1][0].first = j;
50         }
51         graph[j + 1][0].first = (j + 1) + 4;
52         graph[j + 2][0].first = (j + 2) + 4;
53         graph[j + 3][0].first = (j + 3) + 1;
54         graph[j + 4][0].first = (j + 4) - 1;
55         graph[j + 5][0].first = (j + 5) - 4;
56         graph[j + 6][0].first = (j + 6) - 4;
57         graph[j + 7][0].first = (j + 7) + 1; // non-oriented graph
58         graph[(j + 7) + 1][0].first = j + 7; // non-oriented graph
59     } else if (i == 2 - 2 && 2 - euler_number == saddles) { // Last step
60         graph[j][0].first = j - 1;
61         graph[j + 1][0].first = (j + 1) + 4;
62         graph[j + 2][0].first = (j + 2) + 4;
63         graph[j + 3][0].first = (j + 3) + 1;
64         graph[j + 4][0].first = (j + 4) - 1;
65         graph[j + 5][0].first = (j + 5) - 4;
66         graph[j + 6][0].first = (j + 6) - 4;
67         graph[j + 7][0].first = 0;
68     } else {

```

```

68     graph[j][0].first = j - 1;
69     graph[j + 1][0].first = (j + 1) + 4;
70     graph[j + 2][0].first = (j + 2) + 4;
71     graph[j + 3][0].first = (j + 3) + 1;
72     graph[j + 4][0].first = (j + 4) - 1;
73     graph[j + 5][0].first = (j + 5) - 4;
74     graph[j + 6][0].first = (j + 6) - 4;
75     graph[j + 7][0].first = (j + 7) + 1; // non-oriented graph
76     graph[(j + 7) + 1][0].first = j + 7; // non-oriented graph
77 }
78 }
79 graphs.push_back(graph);
80 for (int i = (2 - euler_number) + 1; i < saddles; i += 2) {
81     int j = i * 4;
82     if (i == saddles - 1) {
83         graph[j][0].first = j - 1;
84         graph[j - 1][0].first = j;
85
86         graph[j + 1][0].first = (j + 1) + 2; //equal
87         graph[(j + 1) + 2][0].first = j + 1;
88
89
90         graph[j + 2][0].first = (j + 2) - 4;
91         graph[(j + 2) - 4][0].first = j + 2;
92
93         graph[j + 3][0].first = (j + 3) - 2; // equal
94         graph[(j + 3) - 2][0].first = j + 3;
95     } else {
96         graph[j][0].first = j - 1;
97         graph[j - 1][0].first = j;
98
99         graph[j + 1][0].first = (j + 1) + 4;
100        graph[(j + 1) + 4][0].first = j + 1;
101
102
103        graph[j + 2][0].first = (j + 2) - 4;
104        graph[(j + 2) - 4][0].first = j + 2;
105
106        graph[j + 3][0].first = (j + 3) + 1;
107        graph[(j + 3) + 1][0].first = j + 3;
108    }
109    graphs.push_back(graph);
110 }
111 int fixed_size = graphs.size();
112 for (int i = 0; i < fixed_size; i++) {
113     vector<vector<pair<int, char>>> graph_reverse(0);
114     graph_reverse.assign(graphs[i].begin(), graphs[i].end());
115     for (int j = 0; j < graph_reverse.size(); j++) {

```

```

116     int c = graph_reverse[j][1].first;
117     graph_reverse[j][1].first = graph_reverse[j][2].first;
118     graph_reverse[j][2].first = c;
119 }
120 graphs.push_back(graph_reverse);
121 }
122 return graphs;
123 }

```

10.10 Поиск соседей

```

1 vector<pair<char, vector<int>>> find_neighbors(vector<vector<pair<int, char>>>&
    graph) {
2     vector<pair<char, vector<int>>> cycles(0);
3     // Sources
4     vector<vector<int>> cycles_st = find_cycles(graph, 's', 't');
5     vector<pair<char, vector<int>>> a_cycles(0);
6     for (int i = 0; i < cycles_st.size(); i++) {
7         a_cycles.push_back(make_pair('a', cycles_st[i]));
8     }
9     // Drains
10    vector<vector<int>> cycles_ut = find_cycles(graph, 'u', 't');
11    vector<pair<char, vector<int>>> o_cycles(0);
12    for (int i = 0; i < cycles_ut.size(); i++) {
13        a_cycles.push_back(make_pair('o', cycles_ut[i]));
14    }
15    // Saddles
16    vector<vector<int>> cycles_su = find_cycles(graph, 's', 'u');
17    vector<pair<char, vector<int>>> s_cycles(0);
18    for (int i = 0; i < cycles_su.size(); i++) {
19        a_cycles.push_back(make_pair('s', cycles_su[i]));
20    }
21    // Uniting
22    cycles.insert(cycles.end(), a_cycles.begin(), a_cycles.end());
23    cycles.insert(cycles.end(), o_cycles.begin(), o_cycles.end());
24    cycles.insert(cycles.end(), s_cycles.begin(), s_cycles.end());
25    // Cycles-neighbors finding
26    int alphas = cycles_st.size();
27    int omegas = cycles_ut.size();
28    int sigmas = cycles_su.size();
29    vector<pair<char, vector<int>>> neighbors(0);
30    for (int i = 0; i < cycles.size(); i++) { // O(n^2)
31        vector<int> empty_vector(0);
32        neighbors.push_back(make_pair(cycles[i].first, empty_vector));
33        char i_cycle_type = cycles[i].first;
34        for (int j = 0; j < cycles[i].second.size(); j++) {
35            int a1 = cycles[i].second[j];
36            int a2 = cycles[i].second[(j + 1) % cycles[i].second.size()];

```

```

37     bool is_found = false;
38     for (int k = 0; k < cycles.size(); k++) {
39         char k_cycle_type = cycles[k].first;
40         if (((i_cycle_type == 'a' && k_cycle_type == 's' && graph[a1][1].first ==
a2)) || (i_cycle_type == 'o' && k_cycle_type == 's' && graph[a1][2].first ==
a2) || (i_cycle_type == 's' && ((k_cycle_type == 'a' && graph[a1][1].first ==
a2) || (k_cycle_type == 'o' && graph[a1][2].first == a2)))) {
41             if (k == i) {
42                 continue;
43             }
44             for (int t = 0; t < cycles[k].second.size(); t++) {
45                 int b1 = cycles[k].second[t];
46                 int b2 = cycles[k].second[(t + 1) % cycles[k].second.size()];
47                 if ((a1 == b1 && a2 == b2) || (a1 == b2 && a2 == b1)) {
48                     neighbors[i].second.push_back(k);
49                     is_found == true;
50                     break;
51                 }
52             }
53             if (is_found) {
54                 break;
55             }
56         }
57     }
58     if (cycles[i].second.size() == 2) {
59         break;
60     }
61 }
62 }
63 return neighbors;
64 }

```

10.11 Проверка сепаратрис на пересечение

```

1 bool is_separatres_cross(vector<float> sep1, vector<float> sep2) {
2     float a = sep1[0], a0 = sep1[1], b = sep1[2], b0 = sep1[3], t1 = sep1[4], t2 =
sep1[5];
3     float c = sep2[0], c0 = sep2[1], d = sep2[2], d0 = sep2[3], k1 = sep2[4], k2 =
sep2[5];
4     float max_t = max(t1, t2);
5     float min_t = min(t1, t2);
6     t1 = min_t;
7     t2 = max_t;
8     float max_k = max(k1, k2);
9     float min_k = min(k1, k2);
10    k1 = min_k;
11    k2 = max_k;
12    if ((a0 != c0 || b0 != d0) && a*d == b*c) {

```

```

13     return false; // parallel
14 }
15 if (a0 == c0 && b0 == d0 && a*d == b*c && t2 > k1 && k2 > t1) {
16     return true; // coincide
17 } else if (a0 == c0 && b0 == d0 && a*d == b*c) {
18     return false; // lying on one line
19 }
20 float y_cross = (b * d * c0 - b * c * d0 - b * d * a0 + d * a * b0) / (d * a -
    b * c); // calculated before (linear algebra)
21 float x_cross;
22 float t_cross, k_cross;
23 if (b != 0) {
24     t_cross = (y_cross - b0) / b; // calculated before (linear algebra)
25 } else {
26     x_cross = (c * y_cross - c * d0) / d + c0; // calculated before (linear
    algebra)
27     t_cross = (x_cross - a0) / a; // calculated before (linear algebra)
28 }
29 if (d != 0) {
30     k_cross = (y_cross - d0) / d; // calculated before (linear algebra)
31 } else {
32     x_cross = (a * y_cross - a * b0) / b + a0; // calculated before (linear
    algebra)
33 }
34 bool is_crossing = (t1 < t_cross && t_cross < t2) && (k1 < k_cross && k_cross <
    k2);
35 return is_crossing;
36 }

```

10.12 Поиск сепаратрис 1

```

1 vector<pair<char, vector<float>>> find_sep(int side_len, int center_len, vector<
    pair<char, int>> source_order, vector<pair<char, int>> drain_order, vector<
    pair<char, vector<int>>> neighbors) {
2     vector<pair<int, pair<float, float>>> side(side_len);
3     vector<pair<int, pair<float, float>>> center(center_len);
4     for (int i = 1; i < side_len + 1; i++) {
5         float y = -180 * i / (side_len + 1) + 90;
6         side[i - 1] = (make_pair(source_order[i - 1].second, make_pair(0, y)));
7     }
8     for (int i = 1; i < center_len + 1; i++) {
9         float y = -180 * i / (center_len + 1) + 90;
10        center[i - 1] = (make_pair(drain_order[i - 1].second, make_pair(180, y)));
11    }
12
13    vector<pair<char, vector<float>>> left;
14    vector<pair<char, vector<float>>> right;
15    for (int i = 0; i < side.size(); i++) {

```

```

16     if (i % 2) {
17         vector<float> coords = {side[i - 1].second.first - side[i].second.first,
side[i].second.first, side[i - 1].second.second - side[i].second.second, side[
i].second.second, 0, 1};
18         right.push_back(make_pair('s', coords));
19         coords = {side[i + 1].second.first - side[i].second.first, side[i].second.
first, side[i + 1].second.second - side[i].second.second, side[i].second.
second, 0, 1};
20         right.push_back(make_pair('s', coords));
21     }
22 }
23 for (int i = 0; i < center.size(); i++) {
24     if (i % 2) {
25         vector<float> coords = {center[i - 1].second.first - center[i].second.first
, center[i].second.first, center[i - 1].second.second - center[i].second.
second, center[i].second.second, 0, 1};
26         right.push_back(make_pair('u', coords));
27         coords = {center[i + 1].second.first - center[i].second.first, center[i].
second.first, center[i + 1].second.second - center[i].second.second, center[i
].second.second, 0, 1};
28         right.push_back(make_pair('u', coords));
29     }
30 }
31
32 for (int i = 0; i < source_order.size(); i++) {
33     if (true /*source_order[i].first == 's'*/) {
34         for (int j = 0; j < neighbors[source_order[i].second].second.size(); j++) {
35             if (neighbors[source_order[i].second].second[j] == source_order[(i - 1 +
source_order.size()) % source_order.size()].second || neighbors[source_order[i
].second].second[j] == source_order[(i + 1 + source_order.size()) %
source_order.size()].second) {
36                 continue;
37             }
38             float xi, yi, xj, yj;
39             for (int k = 0; k < side.size(); k++) {
40                 if (side[k].first == source_order[i].second) {
41                     xi = side[k].second.first;
42                     yi = side[k].second.second;
43                     break;
44                 }
45             }
46             for (int k = 0; k < center.size(); k++) {
47                 if (center[k].first == neighbors[source_order[i].second].second[j]) {
48                     xj = center[k].second.first;
49                     yj = center[k].second.second;
50                     break;
51                 }
52             }

```



```

53     vector<float> sep = {xj - xi, xi, yj - yi, yi, 0, 1};
54     bool is_crossing = false;
55     for (int k = 0; k < left.size(); k++) {
56         if (is_separatres_cross(sep, left[k].second)) {
57             is_crossing = true;
58             break;
59         }
60     }
61     if (is_crossing) {
62         sep = {xj - (xi + 360), (xi + 360), yj - yi, yi, 0, 1};
63         if (source_order[i].first == 's') {
64             right.push_back(make_pair('u', sep));
65         } else if (source_order[i].first == 'a') {
66             sep = {xi - xj, xj, yi - yj, yj, 0, 1};
67             right.push_back(make_pair('s', sep));
68         }
69     } else {
70         if (source_order[i].first == 's') {
71             left.push_back(make_pair('u', sep));
72         } else if (source_order[i].first == 'a') {
73             sep = {xi - xj, xj, yi - yj, yj, 0, 1};
74             left.push_back(make_pair('s', sep));
75         }
76     }
77 }
78 }
79 }
80
81 for (int i = 0; i < drain_order.size(); i++) {
82     if (drain_order[i].first == 's') {
83         for (int j = 0; j < neighbors[drain_order[i].second].second.size(); j++) {
84             if (neighbors[drain_order[i].second].second[j] == drain_order[(i - 1 +
85 drain_order.size()) % drain_order.size()].second || neighbors[drain_order[i].
86 second].second[j] == drain_order[(i + 1 + drain_order.size()) % drain_order.
87 size()].second) {
88                 continue;
89             }
90             float xi, yi, xj, yj;
91             for (int k = 0; k < center.size(); k++) {
92                 if (center[k].first == drain_order[i].second) {
93                     xi = center[k].second.first;
94                     yi = center[k].second.second;
95                     break;
96                 }
97             }
98             for (int k = 0; k < side.size(); k++) {
99                 if (side[k].first == neighbors[drain_order[i].second].second[j]) {
100                     xj = side[k].second.first;

```

```

98         yj = side[k].second.second;
99         break;
100     }
101 }
102 vector<float> sep = {xj - xi, xi, yj - yi, yi, 0, 1};
103 bool is_crossing = false;
104 for (int k = 0; k < left.size(); k++) {
105     if (is_separatres_cross(sep, left[k].second)) {
106         is_crossing = true;
107     }
108 }
109 if (is_crossing) {
110     sep = {(xj + 360) - xi, xi, yj - yi, yi, 0, 1};
111     right.push_back(make_pair('s', sep));
112 } else {
113     left.push_back(make_pair('s', sep));
114 }
115 }
116 }
117 }
118 left.insert(left.end(), right.begin(), right.end());
119 return left;
120 }

```

10.13 Поиск сепаратрис 2

```

1 vector<pair<char, vector<float>>> find_separatres_coords(vector<vector<pair<int,
2     char>>>& graph) {
3     vector<pair<char, vector<int>>> neighbors;
4     neighbors = find_neighbors(graph);
5     vector<pair<int, vector<int>>> sources;
6     vector<pair<int, vector<int>>> saddles;
7     vector<pair<int, vector<int>>> drains;
8     // Count sources, saddles and drains number
9     for (int i = 0; i < neighbors.size(); i++) {
10         if (neighbors[i].first == 'a') {
11             sources.push_back(make_pair(i, neighbors[i].second));
12         } else if (neighbors[i].first == 's') {
13             saddles.push_back(make_pair(i, neighbors[i].second));
14         } else if (neighbors[i].first == 'o') {
15             drains.push_back(make_pair(i, neighbors[i].second));
16         }
17     }
18     int sources_num = sources.size();
19     int saddles_num = saddles.size();
20     int drains_num = drains.size();
21     int side_len = 2 * sources_num - 1;
22     int center_len = 2 * drains_num - 1;

```

```

22 vector<pair<char, int>> sourse_order;
23 pair<int, vector<int>> curr_v = sources[0];
24 sourse_order.push_back(make_pair('a', curr_v.first));
25 while (sourse_order.size() != side_len) {
26     for (int i = 0; i < curr_v.second.size(); i++) {
27         bool is_saddle = false;
28         int saddle_index = -1;
29         int saddle_value = -1;
30         for (int j = 0; j < saddles.size(); j++) {
31             if (saddles[j].first == curr_v.second[i]) {
32                 is_saddle = true;
33                 saddle_value = saddles[j].first;
34                 saddle_index = j;
35                 break;
36             }
37         }
38         if (is_saddle) {
39             bool already_was = false;
40             for (int j = 0; j < sourse_order.size(); j++) {
41                 if (saddle_value == sourse_order[j].second) {
42                     already_was = true;
43                 }
44             }
45             if (already_was) {
46                 continue;
47             }
48             int sourse_index1 = -1;
49             int sourse_value1 = -1;
50             int sourse_index2 = -1;
51             int sourse_value2 = -1;
52             for (int t = 0; t < saddles[saddle_index].second.size(); t++) {
53                 // there could be break_factor to speed up cycle
54                 for (int j = 0; j < sources.size(); j++) {
55                     if (sources[j].first == saddles[saddle_index].second[t]) {
56                         if (sourse_index1 == -1) {
57                             sourse_value1 = sources[j].first;
58                             sourse_index1 = j;
59                         } else if (sourse_index2 == -1) {
60                             sourse_value2 = sources[j].first;
61                             sourse_index2 = j;
62                         } else if (sourse_index1 != -1 && sourse_index1 != -1) {
63                             break;
64                         }
65                     }
66                 }
67             }
68             if (sourse_index1 == sourse_index2) {
69                 continue;

```

```

70     }
71     bool is_source_index1_considered = false;
72     bool is_source_index2_considered = false;
73     for (int j = 0; j < source_order.size(); j++) {
74         if (source_order[j].second == source_value1) {
75             is_source_index1_considered = true;
76         }
77         if (source_order[j].second == source_value2) {
78             is_source_index2_considered = true;
79         }
80     }
81     int next_source_value = -1;
82     int next_source_index = -1;
83     if (is_source_index1_considered && is_source_index2_considered) {
84         continue;
85     } else if (is_source_index1_considered && !is_source_index2_considered) {
86         next_source_index = source_index2;
87         next_source_value = source_value2;
88
89     } else if (!is_source_index1_considered && is_source_index2_considered) {
90         next_source_index = source_index1;
91         next_source_value = source_value1;
92     }
93     source_order.push_back(make_pair('s', saddle_value));
94     source_order.push_back(make_pair('a', next_source_value));
95     curr_v = sources[next_source_index];
96 }
97 }
98 }
99 vector<pair<char, int>> drain_order;
100 curr_v = drains[0];
101 drain_order.push_back(make_pair('o', curr_v.first));
102 while (drain_order.size() != center_len) {
103     for (int i = 0; i < curr_v.second.size(); i++) {
104         bool is_saddle = false;
105         int saddle_index = -1;
106         int saddle_value = -1;
107         for (int j = 0; j < saddles.size(); j++) {
108             if (saddles[j].first == curr_v.second[i]) {
109                 is_saddle = true;
110                 saddle_value = saddles[j].first;
111                 saddle_index = j;
112                 break;
113             }
114         }
115         if (is_saddle) {
116             bool already_was = false;
117             for (int j = 0; j < source_order.size(); j++) {

```

```

118         if (saddle_value == source_order[j].second) {
119             already_was = true;
120         }
121     }
122     for (int j = 0; j < drain_order.size(); j++) {
123         if (saddle_value == drain_order[j].second) {
124             already_was = true;
125         }
126     }
127     if (already_was) {
128         continue;
129     }
130     int drain_index1 = -1;
131     int drain_value1 = -1;
132     int drain_index2 = -1;
133     int drain_value2 = -1;
134     for (int t = 0; t < saddles[saddle_index].second.size(); t++) {
135         // there could be break_factor to speed up cycle
136         for (int j = 0; j < drains.size(); j++) {
137             if (drains[j].first == saddles[saddle_index].second[t]) {
138                 if (drain_index1 == -1) {
139                     drain_value1 = drains[j].first;
140                     drain_index1 = j;
141                 } else if (drain_index2 == -1) {
142                     drain_value2 = drains[j].first;
143                     drain_index2 = j;
144                 } else if (drain_index1 != -1 && drain_index1 != -1) {
145                     break;
146                 }
147             }
148         }
149     }
150     if (drain_index1 == drain_index2) {
151         continue;
152     }
153     bool is_drain_index1_considered = false;
154     bool is_drain_index2_considered = false;
155     for (int j = 0; j < drain_order.size(); j++) {
156         if (drain_order[j].second == drain_value1) {
157             is_drain_index1_considered = true;
158         }
159         if (drain_order[j].second == drain_value2) {
160             is_drain_index2_considered = true;
161         }
162     }
163     int next_drain_value = -1;
164     int next_drain_index = -1;
165     if (is_drain_index1_considered && is_drain_index2_considered) {

```

```

166         continue;
167     } else if (is_drain_index1_considered && !is_drain_index2_considered) {
168         next_drain_index = drain_index2;
169         next_drain_value = drain_value2;
170
171     } else if (!is_drain_index1_considered && is_drain_index2_considered) {
172         next_drain_index = drain_index1;
173         next_drain_value = drain_value1;
174     }
175     drain_order.push_back(make_pair('s', saddle_value));
176     drain_order.push_back(make_pair('o', next_drain_value));
177     curr_v = drains[next_drain_index];
178     break;
179 }
180 }
181 }
182 vector<pair<char, vector<float>>> left;
183 left = find_sep(side_len, center_len, source_order, drain_order, neighbors);
184 bool bad_left_side = false;
185 for (int i = 0; i < left.size(); i++) {
186     for (int j = 0; j < left.size(); j++) {
187         if (is_separatres_cross(left[i].second, left[j].second)) {
188             bad_left_side = true;
189             break;
190         }
191     }
192     if (bad_left_side) {
193         break;
194     }
195 }
196 if (bad_left_side) {
197     reverse(drain_order.begin(), drain_order.end());
198     vector<pair<char, vector<float>>> left;
199     left = find_sep(side_len, center_len, source_order, drain_order, neighbors);
200 }
201 return left;
202 }

```

11 Список литературы

- [1] Гринес В.З., Капкаева С.Х., Починка О.В. Трёхцветный граф как полный топологический инвариант для градиентно-подобных диффеоморфизмов поверхностей, Математический сборник, 2014, том 205, номер 10, 19-46 с.
- [2] Патон К., Алгоритм нахождения базы циклов для неориентированного графа, Communications of the ACM 12, 1969, 514-518 с.
- [3] Круглов В.Е., Починка О.В., Топологическая сопряженность градиентноподобных потоков на поверхностях и эффективные алгоритмы ее различения, СМФН, 2022, том 68, выпуск 3, 467–487 с.
- [4] Алексеев В.Е., Таланов В.А., Графы и модели вычислений, Издательство Нижегородского госуниверситета, 2004. 41-73 с.