**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

Кафедра дискретной математики и алгоритмики

**ЗЯЗЮЛЬКИН СЕРГЕЙ ПАВЛОВИЧ**

**НЕПЕРЕСЕКАЮЩИЕСЯ ПОДГРАФЫ В ГЕОМЕТРИЧЕСКИХ ГРАФАХ**

Отчёт по производственной практике по специальности:

1-31 81 09 «Алгоритмы и системы обработки больших объемов информации»

Руководитель практики от кафедры:

кандидат физ.-мат. наук

В.И. Сарванов

Руководитель практики от организации:

кандидат физ.-мат. наук

В.И. Сарванов

Минск, 2019

# Оглавление

[Оглавление 2](#_Toc10067597)

[Введение 4](#_Toc10067598)

[1 основные сведения о проблеме построения непересекающегося остовного дерева в геометрическом графе 5](#_Toc10067599)

[1.1 Геометрические графы 5](#_Toc10067600)

[1.2 Задача построения непересекающегося остовного дерева в геометрическом графе и ее трудоемкость 6](#_Toc10067601)

[1.3 Параметризованные алгоритмы решения задачи построения непересекающегося остовного дерева в геометрическом графе 7](#_Toc10067602)

[1.4 Полиномиальные случаи задачи построения непересекающегося остовного дерева в геометрическом графе 8](#_Toc10067603)

[1.5 Достаточные условия существования непересекающегося остовного дерева в геометрическом графе 9](#_Toc10067604)

[1.6 Достаточные условия отсутствия непересекающегося остовного дерева в геометрическом графе 10](#_Toc10067605)

[1.7 Оптимизационная постановка задачи построения непересекающегося остовного дерева в геометрическом графе 11](#_Toc10067606)

[2 ОСНОВНЫЕ ОПРЕДЕЛЕНИЯ 13](#_Toc10067607)

[2.1 Геометрические графы 13](#_Toc10067608)

[2.2 Матроиды 14](#_Toc10067609)

[3 задача пересечения двух матроидов 15](#_Toc10067610)

[2.3 Алгоритм решения задачи пересечения двух матроидов 15](#_Toc10067611)

[2.4 Программная реализация 18](#_Toc10067612)

[4 ЗАДАЧА построения непересекающегося ациклического подграфа в геометрическом графе 22](#_Toc10067613)

[3.1 Специальные случаи задачи построения непересекающегося остовного дерева в геометрическом графе 22](#_Toc10067614)

[3.2 Алгоритм частичного перебора с отсечениями для решения задачи построения непересекающегося остовного дерева в геометрическом графе 24](#_Toc10067615)

[3.3 Трудоемкость алгоритма частичного перебора с отсечениями для решения задачи построения непересекающегося остовного дерева в геометрическом графе 26](#_Toc10067616)

[3.4 Особенности программной реализации алгоритма частичного перебора с отсечениями для решения задачи построения непересекающегося остовного дерева в геометрическом графе 28](#_Toc10067617)

[3.5 Алгоритм частичного перебора с отсечениями для решения задачи построения наибольшего непересекающегося ациклического подграфа в геометрическом графе 30](#_Toc10067618)

[5 Алгоритм построения непересекающегося остовного дерева в невыпуклом геометрическом графе 32](#_Toc10067619)

[Заключение 36](#_Toc10067620)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 37](#_Toc10067621)

[Приложение А 39](#_Toc10067622)

Введение

Тематика диссертации относится к интенсивно развивающейся области исследований, находящейся «на стыке» теории графов и комбинаторной вычислительной геометрии. Задачи построения непересекающихся подграфов возникают, в частности, в автоматизации проектирования интегральных схем и в робототехнике. Кроме того, они имеют прямое отношение к задачам построения оптимальных по различным критериям плоских триангуляций, играющих ключевую роль в ряде прикладных областей.

Диссертационная работа направлена на исследование и разработку алгоритмов решения задач построения непересекающихся подграфов в геометрических графах. Основное внимание предполагается уделить «элементарным» подграфам (деревьям, простым циклам, паросочетаниям). Планируется разработка алгоритмов построения непересекающихся подграфов в тех классах геометрических графов, для которых доказано существование таких подграфов. Предполагается поиск новых классов геометрических графов, допускающих полиномиальные алгоритмы распознавания и построения элементарных непересекающихся подграфов. Разработанные алгоритмы будут программно реализованы.

За отчетный период необходимо было изучить материал по теме «Полиномиальные случаи задачи построения непересекающегося остовного дерева в геометрическом графе», изучить существующие алгоритмы для решения задачи построения непересекающегося остовного дерева в геометрическом графе, разработать и реализовать точный экспоненциальный алгоритм частичного перебора с отсечениями для решения задачи построения непересекающегося остовного дерева в геометрическом графе, модицировать разработанный алгоритм для решения задачи построения наибольшего непересекающегося ациклического подграфа в геометрическом графе.

основные сведения о проблеме построения непересекающегося остовного дерева в геометрическом графе

1. Геометрические графы

*Геометрическим графом* называется граф, уложенный на плоскости, у которого все ребра − отрезки. Напомним, что под укладкой графа на плоскости подразумевается такое взаимно-однозначное отображение его вершин в точки плоскости, а ребер в дуги (кривые, являющиеся гомеоморфными образами отрезка ), соединяющие эти точки, при котором:

1. никакая дуга не содержит образов вершин, отличных от ее конечных точек;
2. две смежные дуги содержат только одну общую конечную точку;
3. для любых двух несмежных дуг существует не более одной точки, в которой они пересекаются (такое пересечение дуг вне образов вершин называется *собственным*).

Далее всюду будем называть образы вершин и ребер при отображении соответственно вершинами и ребрами соответствующего геометрического графа.

*Индексом пересечения ребра* геометрического графа назовем число собственных пересечений данного ребра. Наибольший индекс пересечения ребра среди всех ребер геометрического графа назовем *индексом пересечения* *геометрического графа*. Геометрический граф, индекс пересечения которого равен нулю, называется *непересекающимся*.

В дальнейшем будем рассматривать только геометрические графы, вершины которых находятся в *общем положении*, т.е. никакие три вершины графа не лежат на одной прямой.

Выпуклую оболочку множества вершин геометрического графа будем называть *выпуклой оболочкой* этого *графа*. Говорят, что вершины геометрического графа находятся в *выпуклом положении*, если они лежат на выпуклой оболочке графа . Такой геометрический граф называют *выпуклым*. Геометрический граф, не являющийся выпуклым, соответственно называют *невыпуклым*. Вершины невыпуклого геометрического графа, лежащие внутри его выпуклой оболочки, называются *внутренними*.

Для краткости изложения непересекающееся остовное дерево будем обозначать через *.*

1. Задача построения непересекающегося остовного дерева в геометрическом графе и ее трудоемкость

Рассмотрим *задачу распознавания в геометрическом графе*:

*Вход*: геометрический граф .

*Вопрос*: существует ли в графе ?

В дальнейшем эту задачу будем называть *задачей распознавания* . Соответственно, задачу построения в геометрическом графе будем называть просто *задачей построения .* В работе [1] представлен следующий важный результат.

**Теорема.** *Пусть – геометрический граф, обладающий следующими свойствами: степень любой вершины не меньше трех, индекс пересечения графа не меньше двух. Задача распознавания в графе является NP-полной.*

Из приведенной теоремы следует, что задача построения является NP-трудной. Тем не менее, эти факты не исключают возможность разработки эффективных приближений с константным коэффициентом аппроксимации. Имеется два естественных способа построения приближения .

Первый способ заключается в построении остовного дерева с минимальным числом пересечений. Обозначим минимальное число пересечений в геометрического графа , увеличенное на единицу, через . О сложности задачи приближения параметра говорит следующая теорема.

**Теорема.** *Пусть − геометрический граф, содержащий пар пересекающихся ребер. Задача приближения параметра с коэффициентом аппроксимации является NP-трудной для любого* .

Второй способ приближения заключается в построении непересекающегося остовного леса с минимальным числом компонент. Обозначим минимальное число компонент в непересекающемся остовном лесе геометрического графа через . Как и в случае параметра , эффективного приближения параметра не существует при условии, что .

**Теорема.** *Пусть − геометрический граф, содержащий пар пересекающихся ребер. Задача приближения параметра с коэффициентом аппроксимации является NP-трудной для любого .*

Таким образом, не только задача построения , но и задача построения эффективного приближения , является NP-трудной.

1. Параметризованные алгоритмы решения задачи построения непересекающегося остовного дерева в геометрическом графе

Одним из направлений разработки эффективных алгоритмов для NP-трудных задач является построение параметризованных алгоритмов. Ключевым моментом при разработке параметризованного алгоритма является выбор подходящего параметра.

Самым простым способом решения задачи построения непересекающегося подграфа специального вида в геометрическом графе является полный перебор всех подграфов заданного графа. Такой перебор может быть осуществлен за экспоненциальное от числа ребер в графе время. Однако, если число пересекающихся пар ребер значительно меньше общего числа ребер в графе и необходимое свойство (в данном случае, что подграф содержит остовное дерево) может быть проверено для непересекающегося подграфа за полиномиальное время, то существует более оптимальная, но все еще наивная стратегия: перебрать все непересекающиеся подграфы, получаемые из исходного графа удалением одного из ребер для каждой пары пересекающихся ребер. Число перебираемых при использовании данного подхода подграфов не превосходит , где – число пар пересекающихся ребер. Таким образом, используя наивный подход, задача построения для геометрического графа может быть решена за время , где -нотация скрывает полиномиальный член.

Первое улучшение наивного подхода было предложено в 2005-ом году. В работе [2] развиты идеи наивного подхода, предложено решение задачи построения в геометрическом графе с трудоемкостью . Несмотря на то, что улучшение являлось незначительным, оно имело теоретический интерес и стимулировало дальнейшее развитие переборных алгоритмов для решения задачи построения .

В работе [3] была предложена идея сведения задачи построения в геометрическом графе к задаче построения в геометрическом графе меньшего размера, при этом должно существовать биективное отображение между множествами графов и . Такое сведение в дальнейшем будем называть *кернелизацией.* Такой граф может быть получен из графа последовательным стягиванием ребер с индексом пересечения, равным . Отметим, что стягивание непересекающихся ребер не нарушает общую структуру пересечений в графе. Легко заметить, что кернелизация может быть выполнена за полиномиальное (на самом деле, линейное) время. Для ряда графов кернелизация может существенно уменьшить размер задачи. Используя идею кернелизации и фиксируя порядок перебора ребер, может быть получен алгоритм решения задачи построения NST, имеющий время работы [3].

**Теорема.** *Пусть − геометрический граф, содержащий пар пересекающихся ребер. За время можно построить NST в графе , если оно существует*.

Задача распознавания может быть решена за полиномиальное время для выпуклых геометрических графов. Этот факт создает предпосылки к созданию параметризованных алгоритмов, использующих в качестве параметра число внутренних вершин геометрического графа.

**Теорема.** *Пусть G − геометрический граф на n вершинах. Задача распознавания NST в графе G может быть решена за время . В качестве параметра выступает число внутренних вершин графа .*

Авторы теоремы отмечают, что соответствующий алгоритм трудно реализовать на практике. В связи с этим они предлагают более «практичный» алгоритм, асимптотика которого, однако, немного хуже.

**Теорема.** *Пусть − геометрический граф на n вершинах. Задача распознавания в графе G может быть решена за время . В качестве параметра выступает число внутренних вершин графа .*

Для решения задачи распознавания также применяется вероятностный подход. Так, в работе [2] представлен следующий результат.

**Теорема.** *Пусть − геометрический граф, содержащий пар пересекающихся ребер. Существует вероятностный алгоритм типа Монте-Карло с односторонней ошибкой для решения задачи распознавания , имеющий вероятность успеха и трудоемкость .*

1. Полиномиальные случаи задачи построения непересекающегося остовного дерева в геометрическом графе

Одним из самых простых случаев геометрического графа, допускающего проверку наличия и построения за полиномиальное время, является непересекающийся геометрический граф. Любой подграф такого геометрического графа будет непересекающимся по определению. Поэтому для построения можно применить любой из алгоритмов построения обычного остовного дерева. Например, может быть использован обычный поиск в глубину или ширину.

Некоторые NP-трудные геометрические задачи на плоскости решаются за полиномиальное время для случая, когда точки находятся в выпуклом положении. Примером такой задачи является задача построения минимальной триангуляции множества точек на плоскости. Не исключением является и задача построения .

**Теорема.** *Пусть – выпуклый геометрический граф на вершинах. Задача построения в графе может быть решена за время с использованием метода динамического программирования [4].*

Еще один полиномиальный случай задачи построения рассмотрен в диссертации [5].

**Теорема.** *Пусть такой геометрический граф, что для произвольной тройки , , его ребер выполняется условие . Тогда задача построения в графе является полиномиально разрешимой.*

Полиномиальная разрешимость данного случая обеспечивается сводимостью к известной полиномиально разрешимой задаче о пересечении двух матроидов. Из данной теоремы следует более простой полиномиальный случай задачи построения .

**Следствие.** *Проблема построения в геометрическом графе с индексом пересечения, равным единице, является полиномиально разрешимой.*

Данное следствие создает предпосылки к созданию параметризованного алгоритма построения , где в качестве параметра выступает число ребер, имеющих не менее двух собственных пересечений.

1. Достаточные условия существования непересекающегося остовного дерева в геометрическом графе

Рассмотрим три вершины , , геометрического графа и образуемый ими треугольник . Отметим, что стороны треугольника могут не являться ребрами графа . Треугольник называют *пустым*, если внутри него не содержится ни одна из вершин графа . Если подграф геометрического графа , порожденный тремя вершинами , , , является несвязным, то треугольник также называют *несвязным*. Число несвязных пустых треугольников в геометрическом графе обозначим через . В работе [6] представлен следующий результат.

**Теорема.** *Если геометрический граф на вершинах, удовлетворяет условию , то в графе существует .*

Напомним, что *дополнением* геометрического графа называется геометрический граф на том же множестве вершин, такой, что две вершины в графе смежны тогда и только тогда, когда они не смежны в графе .

**Теорема.** *Дополнение остовного дерева, индекс пересечения которого больше нуля, содержит NST.*

Случай, когда вместо дополнения остовного дерева рассматривается дополнение , оказывается значительно сложнее. Поэтому он вводится дополнительное ограничение на выпуклость .

**Теорема 10.** *Дополнение , вершины которого находятся в выпуклом положении, содержит тогда и только тогда, когда обладает по крайней мере двумя различными компонентами на границе своей выпуклой оболочки.*

Ряд работ посвящен исследованию локальных достаточных условий существования . В работе [7] была сформулирована следующая гипотеза.

**Гипотеза.** *Пусть – произвольное целое число и – геометрический граф по крайней мере с вершинами. Тогда, если для любого множества из вершин графа подграф, индуцированный множеством , содержит , то и граф содержит .*

Доказательство или опровержение этой гипотезы пока получено не было. Однако в работе [8] приведено подтверждение гипотезы в частном случае, когда .

**Теорема.** *Если для любого подмножества из 6 вершин геометрического графа порядка подграф, индуцированный множеством , содержит , то и сам граф содержит .*

1. Достаточные условия отсутствия непересекающегося остовного дерева в геометрическом графе

Пусть задан полный геометрический граф . Рассмотрим некоторый минимальный подграф графа такой, что любое графа содержит хотя бы одно ребро подграфа . Очевидно, что удаление всех ребер подграфа приводит к тому, что в получаемом графе не существует . Такой подграф будем называть *блокатором NST* или, в дальнейшем, просто *блокатором*. Подробное исследование различных характеристик блокаторов проведено в работе [9]*.* Основными являются следующие результаты.

**Теорема.** *Пусть − полный геометрический граф, а – подграф графа такой, что в графе не существует , диаметр которого не превосходит . Тогда подграф является блокатором.*

В случае, когда на геометрический граф накладываются дополнительные ограничения, в частности ограничение на выпуклость геометрического графа, может быть получен более сильный результат.

**Теорема.** *Пусть – выпуклый полный геометрический граф, а – подграф графа такой, что в графе не существует непересекающееся остовное дерево, диаметр которого не превосходит . Тогда подграф B является блокатором.*

Далее рассмотрим следующую экстремальную проблему для полного геометрического графа: как много произвольных ребер может быть удалено из полного геометрического графа на вершинах, чтобы оставшийся граф по-прежнему содержал . Основные результаты по этой проблеме представлены в работе [10].

**Теорема.** *Для любого , и всех полных геометрических графов на вершинах, а также для всех подграфов графа , содержащих не более вершин, геометрический граф содержит непересекающееся поддерево на вершинах. Данная оценка является строгой относительно числа ребер в подграфе : каждый полный геометрический граф содержит подграф с ребрами такой, что любое непересекающееся поддерево графа содержит не более вершин.*

1. Оптимизационная постановка задачи построения непересекающегося остовного дерева в геометрическом графе

В большинстве приложений геометрических графов наличие пересекающихся ребер является нежелательной характеристикой. Некоторые геометрические структуры, например триангуляция, не содержат пересекающихся ребер по определению. Остовные деревья этим свойством не обладают. Рассмотрим две оптимизационные проблемы, а именно построение минимальной и максимальной длины в полном геометрическом графе. Под длиной подразумевается сумма длин всех ребер, входящих в дерево.

Отсутствие пересекающихся ребер в остовном дереве минимальной длины следует из неравенства треугольника. Таким образом, любое остовное дерево минимальной длины автоматически будет являться непересекающимся. Существует много полиномиальных алгоритмов построения минимальной длины в полном геометрическом графе. Примерами таких алгоритмов являются алгоритм Прима и алгоритм Краскала.

В случае проблемы построения NST максимальной длины ситуация является диаметрально противоположной. Максимизация длины входящих в остовное дерево ребер вступает в противоречие с необходимостью обеспечить отсутствие пересекающихся ребер. Есть предположение, что проблема построения максимальной длины в полном геометрическом графе является NP-трудной, однако доказательство или опровержение этого предположения пока не было получено. В статье [11] произведен анализ данной проблемы и предложенный приближенный алгоритм ее решения.

**Теорема.** *Пусть* *− полный геометрический граф на вершинах. Существует приближенный алгоритм решения задачи построения в геометрическом графе с коэффициентом аппроксимации и трудоемкостью .*

На текущий момент данный алгоритм является лучшим из известных приближений максимальной длины в полном геометрическом графе.

ОСНОВНЫЕ ОПРЕДЕЛЕНИЯ

1. Геометрические графы

*Геометрическим графом* называется граф, уложенный на плоскости, у которого все ребра − отрезки. Напомним, что под укладкой графа на плоскости подразумевается такое взаимно-однозначное отображение его вершин в точки плоскости, а ребер в дуги (кривые, являющиеся гомеоморфными образами отрезка ), соединяющие эти точки, при котором [5]:

1. никакая дуга не содержит образов вершин, отличных от ее конечных точек;
2. две смежные дуги содержат только одну общую конечную точку;
3. для любых двух несмежных дуг существует не более одной точки, в которой они пересекаются (такое пересечение дуг вне образов вершин называется *собственным*).

Далее всюду будем называть образы вершин и ребер при отображении соответственно вершинами и ребрами полученного геометрического графа.

*Индексом пересечения ребра* геометрического графа назовем число собственных пересечений данного ребра. Наибольший индекс пересечения ребра среди всех ребер геометрического графа будем называть *индексом пересечения* *геометрического графа*. Геометрический граф, индекс пересечения которого равен нулю, называется *непересекающимся*.

В дальнейшем будем рассматривать только геометрические графы, вершины которых находятся в *общем положении*, т.е. никакие три вершины графа не лежат на одной прямой.

Выпуклую оболочку множества вершин геометрического графа будем называть *выпуклой оболочкой* этого *графа*. Говорят, что вершины геометрического графа *G* находятся в *выпуклом положении*, если они лежат на выпуклой оболочке графа *G*. Такой геометрический граф *G* называют *выпуклым*. Геометрический граф, не являющийся выпуклым, соответственно называют *невыпуклым*. Вершины невыпуклого геометрического графа, лежащие внутри его выпуклой оболочки, называются *внутренними*.

Для краткости изложения непересекающееся остовное дерево будем обозначать через *NST*.

1. Матроиды

*Матроидом* называют упорядоченную пару , где представляет собой некоторое конечное множество, а – множество подмножеств множества , удовлетворяющее следующим условиям:

1. ;
2. Если и , то ;
3. Если , и , то существует элемент такой, что .

Если – матроид , то называют *матроидом на множестве* . Элементы множества называются *независимыми множествами* матроида , а множество – *носителем* матроида . Множества и матроида часто обозначаются, как и соответственно. Подмножество множества , которое не содержится в множестве , называется *зависимым*. Говорят, что множество является *циклом* матроида , если – минимальное по включению зависимое множество матроида . Максимальные по включению независимые множества матроида называются *базисами* или *базами*.

**Утверждение 1.** Если и – базисы матроида , то .

**Утверждение 2** [3]**.** Пусть – независимое множество матроида , и – зависимое множество. Тогда в множестве содержится единственный цикл матроида , причем этот цикл содержит элемент .

Пусть – геометрический граф. Рассмотрим два примера матроидов на множестве ребер графа .

1. *Матроид циклов* или *графический матроид*. Независимое множество матроида представляет собой ациклическое подмножество ребер графа , цикл – простой цикл графа , а базис – остовный лес графа .
2. *Матроид разбиения*. Пусть задано некоторое разбиение множества ребер графа, т.е. задано семейство непересекающихся подмножеств множества , покрывающих . Подмножество ребер графа называется независимым в том и только в том случае, если никакие два ребра из не лежат в одном и том же множестве разбиения . Циклом матроида разбиения является любое множество, состоящее из двух ребер одного и того же множества разбиения . Множество ребер, содержащее ровно по одному ребру из каждого множества разбиения , является базисом матроида разбиения. Пусть множество ребер графа можно разбить на непересекающиеся подмножества так, что любые два ребра одно и того же подмножества пересекаются между собой, а любые два ребра из разных подмножеств – нет. Будем называть такой матроид *матроидом пересечений*.
3. задача пересечения двух матроидов

Пусть заданы два матроида и , базирующихся на одном и том же множестве элементов . *Задача пересечения двух матроидов* заключается в нахождении такого наибольшего по мощности множества , которое являлось бы независимым для обоих матроидов, т.е. для которого выполняются следующие условия:

1. ;
2. ;
3. ;
4. является наибольшим по мощности множеством среди всех множеств, удовлетворяющих условиям 1−3.

Для краткости изложения задачу пересечения двух матроидов будем называть *задачей* *TMI*.

Задача TMI является полиномиально разрешимой. Алгоритм решения этой задачи будет рассмотрен в следующей подглаве. Следует отметить, что задача пересечения трех матроидов уже является NP-трудной.

1. Алгоритм решения задачи пересечения двух матроидов

Рассмотрим два матроида , и последовательность элементов множества . Говорят, что последовательность является *увеличивающим путем* для некоторого множества , независимого в матроидах и , если применение этой последовательность к множеству переводит это множество в множество , также являющееся независимым в матроидах и , такое, что . Под *применением последовательности* к множеству понимается последовательное добавление в него всех элементов, находящихся на нечетных позициях, и удаление из него всех элементов, находящихся на четных позициях, причем на добавляемые и удаляемые элементы накладываются следующие ограничения: добавляемый элемент должен отсутствовать в множестве, а удаляемый – присутствовать в нем. Отметим, что для того, чтобы выполнялось условие , увеличивающий путь всегда содержит нечетное число элементов.

Идея алгоритма решения задачи TMI заключается в следующем. Пусть известно некоторое множество , являющееся независимым в каждом из пересекаемых матроидов. Для начала можно взять пустое множество, т.к. оно является независимым по определению. Затем, пока это возможно, строятся увеличивающие пути. Полученное в результате увеличений множество и будет искомым.

Открытым остается вопрос построения увеличивающего пути для имеющегося множества . Эта проблема решается построением двудольного орграфа специального вида, вершины которого соответствуют элементам множества , и двух множеств и . Множество содержит элементы (вершины), с которых может начинаться увеличивающий путь, множество – элементы (вершины), которыми может заканчиваться увеличивающий путь. Построение увеличивающего пути сводится к нахождению кратчайшего пути, ведущего из вершины из множества в вершину из множества . Нахождение кратчайшего пути выполняется простой модификацией алгоритма поиска в ширину.

*Алгоритм решения задачи TMI* [1].

*Вход*: два матроида и , базирующихся на одном и том же множестве элементов .

*Выход*: максимальное по мощности множество , являющееся независимым для матроидов и .

1. Множество инициализируется пустым множеством.
2. Выполняем поиск увеличивающего пути для множества .
   1. Выполняем инициализацию структур данных.
      1. Очередь инициализируется пустой очередью. Эта очередь будет использоваться для хранения элементов, с которых может начинаться увеличивающий путь.
      2. Множество инициализируется пустым множеством. В этом множестве будут храниться элементы, которыми может заканчиваться увеличивающий путь.
      3. Список смежности инициализируется пустым списком смежности. Этот список смежности будет использоваться для хранения дуг вспомогательного двудольного орграфа, используемого для поиска увеличивающего пути.
      4. Словарь инициализируется пустым словарем. В этом словаре для элементов будет храниться родительский элемент, т.е. элемент, из которой мы пришли в данный во время поиска увеличивающего пути. Этот словарь нужен для восстановления увеличивающего пути.
   2. Выполняем построение вспомогательного двудольного орграфа.
      1. Итерируемся по всем элементам множества .
      2. Пусть на текущей итерации просматривается элемент . Обозначим множество через .
      3. Если множество является независимым в матроидах и , то увеличивающий путь найден – он состоит из единственного элемента . Переходим к *шагу 3*.
      4. Если множество является независимым только в матроиде , то добавляем элемент в хвост очереди . В противном случае, согласно *утверждению 2,* множество содержит единственный цикл в матроиде . Добавляем в список смежности дуги из элементов цикла в элемент , исключая петлю .
      5. Если множество является независимым только в матроиде , то добавляем элемент в множество . В противном случае, согласно *утверждению 2,* множество содержит единственный цикл в матроиде . Добавляем в список смежности дуги из элемента в элементы цикла , исключая петлю .
   3. Выполняем поиск увеличивающего пути.
      1. Если очередь пуста, то увеличивающий путь для множества не существует. Переходим к *шагу 4*.
      2. Извлекаем элемент из головы очереди .
      3. Итерируемся по всем элементам , в которые ведут дуги из элемента (используем список смежности ).
      4. Если элемент еще не просматривался, т.е. словарь не хранит предка данного элемента, то добавляем в словарь в качестве предка вершины .
      5. Если элемент принадлежит множеству , то увеличивающий путь существует, переходим к восстановлению увеличивающего пути (*шаг 2.4*). В противном случае, увеличивающий путь пока не найден, возвращаемся к *шагу 2.3.1*.
   4. Восстанавливаем увеличивающий путь, итерируюсь по предкам элементов, используя словарь . Увеличивающий путь найден, переходим к *шагу 3*.
3. Если увеличивающий путь найден, то применяем его к множеству и возвращаемся к *шагу 2*.
4. Если увеличивающий путь не найден, то алгоритм завершает свою работу, возвращая в качестве ответа множество .

Трудоемкость данного алгоритма зависит от вида пересекаемых матроидов. В частности, ключевую роль играет трудоемкость алгоритма поиска цикла в зависимом множестве матроида.

**Теорема** [1]**.** Пусть на вход алгоритма решения задачи TMI подаются матроиды и , а задача поиска цикла в зависимом множестве может быть решена за время для каждого из матроидов , . Тогда алгоритм решения задачи TMI корректно решает задачу TMI за время .

1. Программная реализация

Для реализации алгоритма решения задачи TMI был выбран язык программирования *Java*. Реализация алгоритма велась с использованием интегрированной среды разработки *Intellij IDEA*, системы автоматической сборки *Gradle*, библиотеки для модульного тестирования *JUnit*.

Алгоритм решения задачи TMI представлен следующим интерфейсом:

@Immutable

public interface MatroidIntersectionAlgorithm<E> {

@NotNull Set<E> findIntersection(

@NotNull Matroid<E> matroid1,

@NotNull Matroid<E> matroid2)

throws AlgorithmException;

}

Алгоритм был реализован в общем виде без привязки к каким-либо конкретным видам пересекаемых матроидов. В связи с этим для представления матроида был создан следующий интерфейс:

@Immutable

public interface Matroid<E> {

@NotNull @Immutable Set<E> getElements();

Optional<@NotEmpty Set<E>> findCircuit(

@NotNull Set<E> subset)

throws MatroidException;

}

Для представления графа, а также его ребер и вершин, было принято решение вместо использования сторонних библиотек разработать собственную реализацию с целью упрощения работы с графом и оптимизации реализации под нужды разрабатываемого алгоритма.

Интерфейс графа оставляет за конкретной реализацией ответы на следующие вопросы: являются ли ребра направленными, допускаются ли в графе петли, допускаются ли в графе кратные ребра.

// V - vertex ID type

public interface Graph<V> extends Cloneable {

@NotNull @Immutable Set<Vertex<V>> getVertices();

int getVerticesNumber();

@NotNull @Immutable Set<Edge<V>> getEdges();

int getEdgesNumber();

boolean addVertex(@NotNull V id, double x, double y);

boolean addVertex(@NotNull Vertex<V> vertex);

boolean removeVertex(@NotNull V id);

boolean removeVertex(@NotNull Vertex<V> vertex);

@NotNull Vertex<V> vertexOf(

@NotNull V id, double x, double y);

boolean addEdge(

@NotNull V idFrom, double xFrom, double yFrom,

@NotNull V idTo, double xTo, double yTo)

throws GraphException;

boolean addEdge(

@NotNull Vertex<V> from, @NotNull Vertex<V> to)

throws GraphException;

boolean addEdge(

@NotNull Edge<V> edge)

throws GraphException;

boolean removeEdge(@NotNull V from, @NotNull V to);

boolean removeEdge(

@NotNull Vertex<V> from, @NotNull Vertex<V> to);

boolean removeEdge(@NotNull Edge<V> edge);

@NotNull Edge<V> edgeOf(

@NotNull V idFrom, double xFrom, double yFrom,

@NotNull V idTo, double xTo, double yTo);

@NotNull Edge<V> edgeOf(

@NotNull Vertex<V> vertex1,

@NotNull Vertex<V> vertex2);

boolean isDirected();

@NotNull @Immutable Collection<Vertex<V>> getNeighbours(

@NotNull V id);

@NotNull @Immutable Collection<Vertex<V>> getNeighbours(

@NotNull Vertex<V> vertex);

int getIntersectionIndex();

@NotNull Optional<Edge<V>> getMostIntersectingEdge();

boolean isIntersecting();

boolean isConnected();

@NotNull @Immutable List<Graph<V>> getConnectedComponents();

@NotNull Optional<Edge<V>> findBridge();

@NotNull Graph<V> copy();

}

Интерфейс вершин крайне прост, он содержит всего два метода: получение идентификатора вершины (элемента, которому соответствует вершина; как правило, в качества идентификатора вершины используется число) и получение координат вершины на плоскости. На реализацию интерфейса вершины накладывается ограничение: вершины с одинаковым идентификатором должны быть равны между собой.

// Vertices with the same ID must be equal

@Immutable

public interface Vertex<V> {

@NotNull @Immutable V getId();

// (x, y)

@NotNull @Immutable Coordinate getCoordinates();

}

Интерфейс ребра позволяет получить его альтернативные представления: в виде пары вершин, в виде отрезка на плоскости, в виде потока вершин. Как и в случае вершины, на реализацию интерфейса ребра накладывается дополнительное ограничение – ребра между одними и теми же вершинами с сохранением направленности должны быть равны между собой.

// V - vertex ID type

// directed and undirected edges (v1, v2) and (v1, v2) must be equal

// undirected edges (v1, v2) and (v2, v1) must be equal

@Immutable

public interface Edge<V> {

@NotNull ImmutablePair<Vertex<V>, Vertex<V>>

asVerticesPair();

default @NotNull @Immutable Segment asSegment() {…}

default @NotNull Stream<Vertex<V>> asStream() {…}

}

Важно отметить, что реализации интерфейса вершин и ребер должны быть immutable (неизменяемыми), что позволяет безопасно передавать их за пределы классов, которым они принадлежат, а также использовать их в многопоточной среде без какой-либо синхронизации.

Для реализации алгоритма были разработаны следующие имплементации представленных интерфейсов: *SimpleVertex* (вершина, соответствующая точке на плоскости), *SimpleUndirectedEdge* (неориентированное ребро), *UndirectedGraphWithIntersections* (неориентированный граф, не допускающий петли и кратные ребра), *BaseMatroidIntersectionAlgorithm* (реализация алгоритма пересечения двух матроидов, представленного в предыдущей подглаве).

Реализация алгоритма является однопоточной, т.к. в дальнейшем предполагается использовать алгоритм пересечения двух матроидов при разработке алгоритмов для решения задачи построения непересекающихся подграфов в геометрическах графах и выполнять распараллеливание на более высоком уровне.

Полная реализация разработанного алгоритма находится в *приложении A*.

1. ЗАДАЧА построения непересекающегося ациклического подграфа в геометрическом графе

В данной главе будут представлены точные алгоритмы построения NST и наибольшего непересекающегося ациклического подграфа в геометрическом графе с экспоненциальной трудоемкостью, базирующиеся на методе частичного перебора с отсечениями.

При переборе для ребра рассматриваются две возможные ситуации: ребро не входит в искомый подграф, ребро входит в искомый подграф. В первом случае происходит удаление рассматриваемого ребра. Во втором – ребро фиксируется и становится обязательным для включения. Во время фиксирования ребра также происходит удаление ребер, которые не могут входить в искомый подграф вместе с зафиксированным. Такими, например, являются ребра, которые пересекают зафиксированное ребро. Во время перебора для каждого варианта возможны два случая:

1. вариант имеет специальный вид, допускающий проверку наличия и построение искомого подграфа за полиномиальное время;
2. вариант порождает некоторое множество подвариантов, которые необходимо проверить.
3. Специальные случаи задачи построения непересекающегося остовного дерева в геометрическом графе

Очевидно, что несвязный геометрический граф не содержит NST.

Воспользуемся тем свойством, что любой мост графа обязательно входит в остовное дерево этого графа при условии, что оно существует. Таким образом, если геометрический граф содержит один или более мостов, то задача построения NST может быть разбита на подзадачи меньшего размера: удалить из геометрического графа все мосты и ребра, пересекающие мосты, и решить задачу построения NST для каждой из полученных компонент связности. Если хотя бы одна из компонент связности не содержит NST, то и исходный геометрический граф не содержит NST. Если для каждой компоненты связности удалось построить NST, то NST исходного геометрического графа может быть получено добавлением удаленных мостов к построенным NST.

Одним из самых простых случаев геометрического графа, допускающего проверку наличия и построения NST за полиномиальное время, является непересекающийся геометрический граф. Любой подграф такого геометрического графа будет непересекающимся по определению. Поэтому для построения NST можно применить любой из алгоритмов построения обычного остовного дерева. Например, может быть использован обычный поиск в глубину или ширину. Большинство алгоритмов построения остовного дерева могут быть легко модифицированы таким образом, чтобы поддерживать заранее заданное множество ребер, которое должно входить в остовное дерево.

Напомним, что задача TMI может быть решена за полиномиальное время. Поэтому, если можно построить матроид пересечений на множестве ребер геометрического графа, то можно проверить наличие и построить NST за полиномиальное время, построив пересечение двух матроидов: графического матроида и матроида пересечений. Если мощность множества ребер, полученного в результате пересечения матроидов, на единицу меньше числа вершин в геометрическом графе, то полученное множество ребер является NST. В противном случае, геометрический граф не содержит NST. Примером геометрического графа, допускающего построение матроида пересечений на множестве ребер, является геометрический граф, имеющий индекс пересечения, равный единице.

Попытка применить алгоритм решения задачи TMI в явном виде для разрабатываемого алгоритма частичного перебора с отсечениями сталкивается со следующей проблемой: к моменту, когда можно применить алгоритм решения задачи TMI, часть ребер геометрического графа уже зафиксирована и обязана входить в NST, однако алгоритм решения задачи TMI в чистом виде не позволяет указать ребра, которые обязательно должны входить в пересечение двух матроидов.

Пусть задан матроид и независимое множество этого матроида. Рассмотрим упорядоченную пару , где , . Докажем, что упорядоченная пара также является матроидом. Для этого проверим, что она удовлетворяет каждому из трех условий из определения матроида. Доказательство будем производить методом «от противного».

1. Пусть . Тогда . Противоречие с тем, что является независимым множеством матроида по определению.
2. Пусть , и . Обозначим , . Из определения упорядоченной пары следует, что , и . Получаем, что матроид не удовлетворяет второму условия из определения матроида – противоречие.
3. Пусть , , и не существует элемент такой, что . Обозначим , . Из определения упорядоченной пары следует, что , , и не существует элемент такой, что . Получаем, что матроид не удовлетворяет третьему условию из определения матроида – противоречие.

Таким образом, упорядоченная пара , полученная из матроида путем фиксирования независимого множества , также является матроидом. Такой матроид будем называть *матроидом с зафиксированным множеством* . Заметим, что каждому независимому множеству матроида соответствует независимое множество матроида .

Приведенная ранее проблема наличия фиксированного множества ребер может быть решена следующим образом. Пусть задан графический матроид , матроид пересечений и зафиксированное множество ребер такое, что и . Используя алгоритм решения задачи TMI, построим пересечение матроидов и , полученных из матроидов и соответственно путем фиксирования независимого множества . Добавив к полученному пересечению множество , получим пересечение исходных матроидов и , причем это пересечение содержит фиксированное множество ребер , что нам и требовалось.

1. Алгоритм частичного перебора с отсечениями для решения задачи построения непересекающегося остовного дерева в геометрическом графе

Напомним, что задача построения NST может быть решена за полиномиальное время для геометрического графа с индексом пересечения, равным 1, за счет сведения к задаче TMI. Используя этот факт и специальные случаи задачи построения NST из предыдущей подглавы, может быть разработан следующий точный экспоненциальный алгоритм частичного перебора с отсечениями для решения задачи NST.

*Алгоритм частичного перебора с отсечениями для решения задачи построения NST.*

*Вход:* геометрический граф , опциональное множество фиксированных ребер .

*Выход:* NST, если оно существует в геометрическом графе .

1. Множество фиксированных ребер инициализируется пустым множеством, если оно не было подано на вход.
2. Очередь задач инициализируется очередью из одного элемента – пары . Очередь будет использоваться для хранения подзадач, которые необходимо решить – пар (подграф , множество зафиксированных ребер подграфа ).
3. Если очередь пуста, то в геометрическом графе отсутствует NST. Алгоритм завершает свою работу.
4. Извлекаем из очереди пару .
5. Если геометрический граф является несвязным, то он не содержит NST, возвращаемся на *шаг 3*.
6. Если геометрический граф является непересекающимся, то выполняем построение NST с фиксированным множеством ребер , используя любую модификацию алгоритма поиска обычного остовного дерева, поддерживающую задание ребер, которые обязаны входить в остовное дерево. Если удалось построить NST, то алгоритм завершает свою работу, возвращая построенное NST в качестве ответа. Если построить NST не удалось, то возвращаемся на *шаг 3*.
7. Если возможно построить матроид пересечений на множестве ребер геометрического графа (например, индекс пересечения графа равен 1), то выполняем построение NST с фиксированным множеством ребер , используя алгоритм решения задачи TMI. Если удалось построить NST, то алгоритм завершает свою работу, возвращая построенное NST в качестве ответа. Если построить NST не удалось, то возвращаемся на *шаг 3*.
8. *Опциональный шаг.* Если геометрический граф содержит хотя бы один мост, то решаем задачу построения NST с указанием соответствующих фиксированных ребер для каждой из компонент связности, которые получаются из графа путем удаления мостов и всех ребер, пересекающих мосты. Если для каждой из компонент связности удалось построить NST, то строим NST графа , объединяя NST компонент связности с мостами графа . Алгоритм завершает свою работу, возвращая построенное NST в качестве ответа. Если хотя бы для одной компоненты связности NST построить не удалось, то возвращаемся на *шаг 3*. Отметим, что решать задачу построения NST для компонент связности можно двумя способами. Первый (самый простой) способ заключается в рекурсивном вызове данного алгоритма для каждой из компонент связности. Второй способ, который может быть полезен для параллельной реализации алгоритма, предполагает добавление в очередь подзадач , где – -ая компонента связности графа , . При этом необходимо модифицировать соответствующим образом данный алгоритм для обработки таких подзадач и выполнения агрегирования результатов после их решения.
9. Выбираем некоторое ребро графа , которое имеет индекс пересечения, больший 0. Такое ребро всегда можно выбрать, т.к. в противном случае алгоритм остановился бы на *шаге 6*. Алгоритм выбора ребра допускает вариации и зависит от конкретной реализации алгоритма. Например, можно брать ребро с наибольшим индексом пересечения или случайное ребро.
10. Если ребро не образует цикл с зафиксированными ребрами , то строим граф , удаляя из графа все ребра, что пересекают ребро , и добавляем в очередь подзадачу .
11. Строим граф удалением ребра и добавляем в очередь подзадачу .
12. Возвращаемся на *шаг 3*.

Представленный алгоритм оставляет за реализацией решение следующих вопросов:

1. Реализация очереди . Задачи могут извлекаться в порядке поступления, случайным образом, в соответствии с некоторым приоритетом, иным образом.
2. Конкретный алгоритм поиска NST с фиксированным множеством ребер, используемый на *шаге 6*.
3. Включение в алгоритм *шага 8*.
4. Алгоритм выбора ребра на *шаге 9*.
5. Распараллеливание алгоритма: пары могут обрабатываться независимо друг от друга.
6. Мемоизация и кеширование с целью исключения повторных вычислений.
7. Трудоемкость алгоритма частичного перебора с отсечениями для решения задачи построения непересекающегося остовного дерева в геометрическом графе

Рассмотрим трудоемкость представленного алгоритма при следующих условиях: произвольная реализация очереди , любой полиномиальный алгоритм поиска NST для *шага 6*, отсутствие *шага 8*, выбор ребра с наибольшим индексом пересечения на *шаге 9*.

Пусть на вход алгоритму подан граф , содержащий пар пересекающихся ребер, время работы *шагов 5-7* алгоритма составляет , время работы *шагов 9-11* – , где , , и – некоторые полиномы.

Подзадачи будем называть *простыми*, если они могут быть решены без дальнейшего разбиения на подзадачи, т.е. будут обработаны на *шагах 5-7*. Подзадачи, не являющиеся простыми, будем называть *составными*. Отметим, что обработка составных задач выполняется на *шагах 5-7* и *шагах 9-11*. Под *размером подзадачи* будем понимать число пар пересекающихся ребер в графе . Граф будем называть *графом подзадачи*. Обработка простых подзадач может быть произведена за время , составных – за время .

Очевидно, что подзадачи размера 0 являются простыми, т.к. графы подзадач являются непересекающимися. Простыми также являются подзадачи , где граф имеет индекс пересечения, равный 1, т.к. такие подзадачи будут обработаны на *шаге 6*. Следовательно, простыми также являются подзадачи размера 1, потому что индекс пересечения графов таких подзадач будет тоже равен 1.

Подзадача либо является простой, либо разбивается на не более чем две подзадачи, каждая из которых, в свою очередь, может быть простой или составной. Заметим, что размер подзадач, образуемых при разбиение составной задачи, не менее чем на 2 меньше размера исходной составной подзадачи. Действительно, если подзадача является составной, то ее индекс пересечения равен минимум 2. Отсюда следует, что и индекс пересечения ребра , выбираемого на *шаге 9*, также будет не меньше 2. При удалении или фиксировании ребра размер получаемой подзадачи уменьшается на число пересекаемых ребром ребер, т.е. не менее чем на 2.

Исходная задача имеет размер . Исходя из того, что при разбиении размер образуемых подзадач не менее чем на 2 меньше размера исходной составной подзадачи, а также учитывая тот факт, что число образуемых подзадач не превосходит 2, получаем, что задача размера может породить не более чем простых подзадач и не более чем составных подзадач. Следовательно, общее время работы алгоритма составляет . Отметим, что полученный алгоритм значительно улучшает предыдущую оценку времени работы , представленную в работе [2].

Влияние *шага 8* на время работы алгоритма является объектом для дальнейшего исследования. С одной стороны, разбиение на подзадачи по компонентам связности после удаления мостов и пересекающих мосты ребер может увеличивать общее число подзадач. И число образуемых простых подзадач, и число образуемых составных подзадач может превышать . В то же время, размеры (число ребер и вершин) графов подзадач могут быть значительно меньше, что потенциально ускоряет обработку подзадач и может приводить к меньшему итоговому времени работы алгоритма.

1. Особенности программной реализации алгоритма частичного перебора с отсечениями для решения задачи построения непересекающегося остовного дерева в геометрическом графе

Для разработки алгоритма частичного перебора с отсечениями для решения задачи построения NST использовалась та же программная среда, что и для решения задачи TMI (см. подглаву 2.2).

Алгоритм решения задачи построения NST представлен следующим интерфейсом:

@Immutable

public interface NcstAlgorithm<V> {

Optional<Graph<V>> findNcst(

final @NotNull Graph<V> graph)

throws AlgorithmException;

}

Алгоритм был реализован в следующем варианте: для хранения подзадач используется несколько потокобезопасных очередей с использованием work-stealing алгоритма извлечения подзадач из очереди; на *шаге 6* используется алгоритм построения остовного дерева, основанный на системе непересекающихся множеств и работающий за время , где – число ребер в графе; *шаг 8* включен в реализацию, причем подзадачи решаются не рекурсивно, а с использованием общих с другими подзадачами очередей и дополнительных механизмов синхронизации; на *шаге 9* выбирается ребро с наибольшим индексом пересечения; реализация является параллельной, причем распараллеливание возможно на произвольное число потоков, не превышающее общее число образуемых подзадач; мемоизация и кеширование не производятся, однако используется ряд вспомогательных классов, позволяющих ускорить некоторые вычисления; на *шаге 7* лишь проверяется, имеет ли граф подзадачи индекс пересечения, равный 1, т.к. полная проверка, можно ли построить матроид пересечений на множестве ребер графа подзадачи, слишком трудоемка.

Параллельная реализация алгоритма основана на общих очередях подзадач и использовании ForkJoinPool-а. Для упрощения реализации и повышения уровня параллелизации между подзадачами не делается различия, т.е. подзадачи, формируемые на *шагах 8*, *10* и *11*, хранятся в одних и тех же очередях подзадач и обрабатываются единообразно. ForkJoinPool представляет собой реализацию пула потоков из стандартной библиотеки Java. Его ключевой особенностью являются эффективные механизмы порождения дочерних подзадач и ожидания завершения их обработки для агрегирования результатов, а также пониженное по сравнению с другими стандартными реализациями пула потоков потребление ресурсов процессора при решении задач, порождающих большое число подзадач. Такая эффективность ForkJoinPool-а обеспечивается за счет использования work-stealing механизма обработки подзадач: «простаивающие» потоки «воруют» (отсюда и название «work-stealing») подзадачи, порожденные подзадачами, обрабатываемыми другими потоками. Это позволяет одновременно обрабатывать число подзадач, значительно превышающее число потоков в ForkJoinPool-е.

С целью уменьшения объема используемой алгоритмом памяти, а также уменьшения числа механизмов синхронизации, многие объекты являются immutable. Таковыми, например, являются реализации интерфейсов *Edge*, *Vertex*, *Matroid*. Это позволяет передавать и использовать эти объекты сразу несколькими потоками без применения механизмов синхронизации. К тому же, это позволяет использовать одни и те же immutable объекты в различных подзадачах (например, одни и те же инстансы вершин графа во всех графах подзадач), что избавляет от необходимости клонировать эти объекты и уменьшает объем потребляемой памяти.

Кеширование и мемоизация в явном виде не применяются, однако разработан и используется ряд вспомогательных классов, позволяющих ускорить некоторые вычисления, в том числе за счет исключения повторных вычислений. Например, для множества фиксированных ребер поддерживается система непересекающихся множеств, представляющая разбиение вершин на компоненты связности и позволяющая быстро ответить на вопрос, приводит ли добавление нового фиксированного ребра к образованию цикла.

Реализованный алгоритм покрыт unit-тестами.

Программный код реализованного алгоритма находится в *Приложении А*.

1. Алгоритм частичного перебора с отсечениями для решения задачи построения наибольшего непересекающегося ациклического подграфа в геометрическом графе

Под *наибольшим подграфом* будем понимать подграф, имеющий наибольшее число ребер. Алгоритм частичного перебора с отсечениями для решения задачи построения NST может быть адаптирован для решения задачи построения наибольшего непересекающегося ациклического подграфа в геометрическом графе.

Алгоритмы построения NST для простых подзадач могут быть легко модифицированы для построения наибольшего ациклического подграфа. Случай несвязного подграфа, в котором явно отсутствует NST, может содержать наибольший ациклический подграф, поэтому тоже должен обрабатываться. Следовательно, отбрасывать его, как это делается в алгоритме решения задачи построения NST, нельзя. В отличие от NST наибольший непересекающийся ациклический подграф может не содержать мосты, поэтому *шаг 8* алгоритма решения задачи построения NST неприменим. Также стоит отметить, что ранняя остановка перебора может быть осуществлена только в случае нахождения NST, т.к. в противном случае не гарантируется, что оставшиеся для перебора варианты не содержат большего непересекающегося ациклического подграфа.

*Алгоритм частичного перебора с отсечениями для решения задачи построения наибольшего непересекающегося ациклического подграфа.*

*Вход:* геометрический граф , опциональное множество фиксированных ребер .

*Выход:* наибольший непересекающийся ациклический подграф графа .

1. Множество фиксированных ребер инициализируется пустым множеством, если оно не было подано на вход.
2. Наибольший ациклический непересекающийся подграф инициализируется графом .
3. Очередь задач инициализируется очередью из одного элемента – пары . Очередь будет использоваться для хранения подзадач, которые необходимо решить – пар (подграф , множество зафиксированных ребер подграфа ).
4. Если очередь пуста или является NST, то алгоритм завершает свою работу, возвращая в качестве ответа.
5. Извлекаем из очереди пару .
6. Если геометрический граф является непересекающимся, то выполняем построение наибольшего непересекающегося ациклического подграфа с фиксированным множеством ребер . Если подграф больше подграфа , то . Переходим на *шаг 4*.
7. Если возможно построить матроид пересечений на множестве ребер геометрического графа (например, индекс пересечения графа равен 1), то выполняем построение наибольшего непересекающегося ациклического подграфа с фиксированным множеством ребер , используя алгоритм решения задачи TMI. Если подграф больше подграфа , то . Переходим на *шаг 4*.
8. Выбираем некоторое ребро графа , которое имеет индекс пересечения, больший 0. Такое ребро всегда можно выбрать, т.к. в противном случае алгоритм остановился бы на *шаге 6*. Алгоритм выбора ребра допускает вариации и зависит от конкретной реализации алгоритма. Например, можно брать ребро с наибольшим индексом пересечения или случайное ребро.
9. Если ребро не образует цикл с зафиксированными ребрами , то строим граф , удаляя из графа все ребра, что пересекают ребро , и добавляем в очередь подзадачу .
10. Строим граф удалением ребра и добавляем в очередь подзадачу .
11. Возвращаемся на *шаг 4*.

Разработанный алгоритм оставляет за реализацией решение следующих вопросов:

1. Реализация очереди . Задачи могут извлекаться в порядке поступления, случайным образом, в соответствии с некоторым приоритетом, иным образом.
2. Конкретный алгоритм поиска наибольшего ациклического непересекающегося подграфа с фиксированным множеством ребер, используемый на *шаге 6*.
3. Алгоритм выбора ребра на *шаге 8*.
4. Распараллеливание алгоритма: пары могут обрабатываться независимо друг от друга.
5. Мемоизация и кеширование с целью исключения повторных вычислений.

Алгоритм решения задачи построения наибольшего ациклического непересекающегося подграфа в геометрическом графе имеет ту же трудоемкость , что и алгоритм решения задачи построения NST.

1. Алгоритм построения непересекающегося остовного дерева в невыпуклом геометрическом графе

Напомним, что задача построения NCST в выпуклом геометрическом графе может быть решена за время при помощи техники динамического программирования. Покажем, что решение задачи построения NCST за время существует и для более общего случая.

*Многоугольником* называется часть плоскости, ограниченная замкнутой ломаной без самопересечений. Эту ломаную называют *границей многоугольника*. Вершины ломаной называются *вершинами многоугольника*, а отрезки – *сторонами многоугольниками*. Отрезки, соединяющие несмежные вершины многоугольника, называют *диагоналями*. Многоугольник называется *выпуклым*, если он лежит по одну сторону от любой прямой, содержащей его сторону (т.е. продолжения сторон многоугольника не пересекают других его сторон). В противном случае многоугольник называется *невыпуклым*. Диагональ многоугольника будем называть *допустимой*, если она лежит строго внутри данного многоугольника. В противном случае диагональ будем называть *недопустимой*. Очевидно, что все диагонали выпуклого многоугольника являются допустимыми.

Рассмотрим геометрический граф, вершины которого являются вершинами некоторого невыпуклого многоугольника, при этом все ребра геометрического графа лежат внутри или на границе этого многоугольника. Такой геометрический граф будем называть *невыпуклым*. Очевидно, что для выпуклого геометрического графа, содержащего не менее 3 вершин, существует аналогичный выпуклый многоугольник. Соответствующий графу многоугольник будем называть *многоугольником* этого *графа*. Не трудно заметить, что геометрический граф не может одновременно являться выпуклым и невыпуклым.

*Триангуляцией* *многоугольника* называется такой геометрический граф, что:

1. каждая внутренняя грань ограничена треугольником;
2. граница внешней грани совпадает с границей триангулируемого многоугольника.

Заметим, что если в выпуклом или невыпуклом геометрическом графе существует NCST, то существует и триангуляция многоугольника этого графа, для которой NCST является подграфом. Таким образом, перебрав все триангуляции многоугольника геометрического графа, можно проверить существование NCST в этом графе.

Рассмотрим некоторый выпуклый или невыпуклый геометрический граф , , и соответствующий ему многоугольник . Обозначим через множество сторон многоугольника . Пусть – множество всех триангуляций многоугольника .

Очевидно, что каждая сторона многоугольника содержится ровно в одной внутренней грани для каждой триангуляции из множества . Пронумеруем вершины графа числами от до вдоль границы многоугольника . Рассмотрим произвольную сторону этого многоугольника. Не нарушая общности, пусть это будет сторона с вершинами и . Проанализируем все треугольники, лежащие внутри многоугольника и содержащие сторону . Таковым является множество треугольников , содержащее треугольники , где , а диагонали и являются допустимыми. Пусть – множество триангуляций из , содержащих . Не трудно заметить, что , .

Зафиксируем некоторый треугольник . Он разбивает множество вершин графа на два множества и , . Рассмотрим два подграфа и геометрического графа , индуцируемые множествами вершин и соответственно. Заметим, что каждый из подграфов , является или геометрическим графом на двух вершинах, или геометрическим графом, которому соответствует некоторый многоугольник, т.е. графом, допускающим дальнейшее аналогичное разбиение. Возьмем произвольные непересекающиеся остовные леса и в подграфах и соответственно. Непересекающийся остовный лес графа получается из лесов и объединением деревьев, содержащих вершину , а также вершины и соответственно при условии наличия ребра в геометрическом графе . В отношении подграфов и возможны следующие случаи:

1. В подграфах , существуют NCST , соответственно. Очевидно, что тогда и графы , содержат NCST ;
2. Один из подграфов содержит NCST , а другой – такой непересекающийся остовный лес на двух деревьях , что вершины , или , принадлежат разным деревьям. Если в графе существует ребро , то и в графе , и в графе существует NCST . В противном случае в графе существует такой непересекающийся остовный лес на двух деревьях , что вершины , принадлежат разным деревьям.
3. Если подграфы , не удовлетворяют условиям 1 и 2, то не трудно заметить, что в графе не существует NCST. Более того, в графе также не существует такого непересекающегося остовного леса на двух деревьях, что вершины и принадлежат разным деревьям.

Таким образом, существование NCST в геометрическом графе может быть проверено следующим образом. Фиксируется некоторая сторона многоугольника . Перебираются все допустимые треугольники на множестве вершин , содержащие сторону . Каждый треугольник разбивает граф на два подграфа , , для которых задача решается рекурсивно. Для тривиального случая, когда геометрический граф содержит ровно две вершины, существование NCST или непересекающегося остовного дерева на двух деревьях определяется наличием ребра между этими двумя вершинами. Если для хоты бы одного разбиения существует NCST, то NCST существует и для графа . В противном случае не существует триангуляции многоугольника , содержащей NCST геометрического графа в качестве подграфа, а значит, геометрическом граф не содержит NCST.

Для выпуклого или невыпуклого геометрического графа задача построения NCST может быть решена с использованием техники динамического программирования. Рассмотрим матрицу размера . Элементу , поставим в соответствие подграф графа , индуцируемый множеством вершин с номерами . Если диагональ является допустимой, то в будем хранить , если подграф содержит NCST; , если подграф содержит такой непересекающийся остовный лес на двух деревьях, что вершины с номерами и принадлежат разным деревьям; в противном случае. Если диагональ является недопустимой, то в будем хранить На первом, восходящем этапе алгоритма производится последовательное заполнение матрицы . Затем проверяется существование NCST в графе . NCST в графе существует тогда и только тогда, когда . Если NCST существует, то оно может быть восстановлено на втором, нисходящем этапе алгоритма, предполагающем рекурсивно восстанавливать NCST путем пробега по матрице и определения разбиений, приводящих к построению NCST.

*Алгоритм построения NCST в выпуклом или невыпуклом геометрическом графе.*

*Вход:* выпуклый или невыпуклый геометрический граф , , и многоугольник графа .

*Выход:* NCST графа , если оно существует.

1. Выполняется нумерация вершин геометрического графа числами от 1 до вдоль границы многоугольника.
2. Если геометрический граф является невыпуклым, то определяем для каждой диагонали многоугольника , является ли она допустимой. Для выпуклого геометрического графа все диагонали являются допустимыми.
3. Выполняем шаги в цикле по от до .
4. Если , то . Иначе .
5. Выполняем шаг в цикле по от до .
6. Если диагональ является недопустимой, то . Иначе , где , принимает значение , если хотя бы одно из значение , равно ; , если или и (, или , ); 1, если и (, или , ); 0 в противном случае.
7. Если , то в геометрическом графе отсутствует NCST. Алгоритм завершает свою работу.
8. Изначально множество ребер NCST пусто: .
9. , .
10. Если и , то .
11. Если , то находим , где определена на шаге . Если и (, или , ), то . Повторяем шаги для , и , .
12. Возвращается в качестве ответа.

Оценим трудоемкость представленного алгоритма. Нумерация вершин геометрического графа на шаге может быть выполнена за время . Является ли геометрический граф выпуклым, можно также определить за время . Трудоемкость проверки всех диагоналей графа на допустимость составляет для невыпуклого геометрического графа. Таким образом, шаг выполняется за время для выпуклого геометрического графа, – для невыпуклого. Если информация о выпуклости или невыпуклости заданного графа подается на вход алгоритму, то для выпуклого геометрического графа шаг будет выполнен за время Шаги представляют собой 3 вложенных цикла по вершинам графа и имеют трудоемкость . Шаги и 12 являются элементарными и занимают время . Не трудно заметить, что шаги и будут вызываться раз каждый. Трудоемкость шага составляет , шага – . Отметим, что трудоемкость шага можно уменьшить до , если предпросчитывать и сохранять на шаге . При этом, правда, потребуется дополнительной памяти для хранения для всех пар , .

Итого, трудоемкость алгоритма построения NCST в выпуклом или невыпуклом геометрическом графе составляет .

Заключение

В соответствии с планом работ за отчетный период было проделано следующее:

* изучен материал по теме «Полиномиальные случаи задачи построения непересекающегося остовного дерева в геометрическом графе»;
* изучены существующие алгоритмы для решения задачи построения непересекающегося остовного дерева в геометрическом графе;
* разработан и программно реализован точный экспоненциальный алгоритм частичного перебора с отсечениями для решения задачи построения непересекающегося остовного дерева в геометрическом графе;
* вычислена и доказана трудоемкость разработанного алгоритма;
* разработана модифицированная версия алгоритма для решения задачи построения наибольшего непересекающегося ациклического подграфа в геометрическом графе.

Дальнейшая работа будет направлена на доработку разработанного алгоритма для решения задачи построения непересекающегося остовного дерева в геометрическом графе, программную реализацию модифицированной версии алгоритма для решения задачи построения наибольшего непересекающегося ациклического подграфа в геометрическом графе, проведение вычислительных экспериментов, исследование задачи построения непересекающегося остовного дерева в геометрическом графе, вершины которого лежат на границе некоторого многоугольника (возможно, невыпуклого), а ребра – внутри или на границе этого многоугольника.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. K. Jansen, G. J. Woeginger. The complexity of detecting crossingfree configurations in the plane. BIT, pp.580–595, 1993.
2. Knauer, C. Configurations with few crossings in topological graphs / C. Knauer, E. Schramm, A. Spillner, A. Wolff // International Symposium on Algorithms and Computations (ISAAC), Springer, 2005. – Springer, 2005. – v. 3827. − P. 604−613.
3. Halldorsson, M. Parameterized algorithms and complexity of non-crossing spanning trees / M. Halldorsson, C. Knauer, A. Spillner, T. Tokuyama // 10th Workshop on Algorithms and Data Structures (WADS), Halifax, Canada, 2007 – Halifax, 2007. – P. 410−421.
4. Parameterized Algorithms and Complexity of Non-Crossing Spanning Trees.
5. Бенедиктович, В.И. Алгоритмы и сложность построения некоторых комбинаторно-геометрических конфигураций: дис. … канд. физ.-мат. наук: 01.01.09 / В.И. Бенедиктович. – Минск, 2001. – 97 с.
6. Rivera-Campo, E. A sufficient condition for the existence of plane spanning trees on geometric graphs / E. Rivera-Campo, V. Urrutia-Galicia // Computational Geometry: Theory and Applications. – 2013. – № 1. – P. 1–6.
7. E. Rivera-Campo. Proceedings of the Japanese Conference on Discrete and Computational Geometry. Tokyo, Japan. December 1998, LNCS. 2000. vol. 1763. pp.274-277.
8. Бенедиктович, В.И. Локальный признак существования плоского остовного дерева в геометрическом графе / В.И. Бенедиктович // Известия Национальной академии наук Беларуси. Серия Физико-математических наук. – 2014. − № 2. – С. 58−63.
9. C. Keller, M. A. Perles, E. Rivera-Campo and V. Urrutia-Galicia, [Blockers for non-crossing spanning trees in complete geometric graphs](http://arxiv.org/abs/1201.4782), in: J. Pach (ed.), Thirty Essays on Geometric Graph Theory, Springer-Verlag, 2013, pp.383-398.
10. O. Aichholzer, S. Cabello, R. Fabila-Monroy, D. Flores-Peñaloza, T. Hackl, et al. Edge-Removal and Non-Crossing Configurations in Geometric Graphs. Discrete Mathematics and Theoretical Computer Science, DMTCS, 2010, 12 (1), pp.75-86.
11. N. Alon, S. Rajagopalan, S. Suri: Long non-crossing configurations in the plane, Fundamenta Informaticae 22 (1995), pp.385–394.
12. Knauer, C. Fixed-parameter algorithms for finding crossing-free spanning trees in geometric graphs / C. Knauer, A. Spillner. – Department of Computer Science, Friedrich-Schiller-Universitat Jena, 2006. – 11 p.
13. Пападимитриу, Х. Комбинаторная оптимизация: Алгоритмы и сложность / Х. Пападимитриу, К. Стайглиц. − М.: Мир, 1984. – 510 с.
14. Oxley, J.G. Matroid Theory / J.G. Oxley. – Oxford University Press, 2006. – 532 p.
15. Бенедиктович, В.И. Непересекающееся остовное дерево геометрического дополнения остовного дерева / В.И. Бенедиктович // Известия Национальной академии наук Беларуси. Серия Физико-математических наук. – 2010. − № 1. – С. 103−109.

Приложение А

Реализация разработанных алгоритмов

// V - vertex ID type

// directed and undirected edges (v1, v2) and (v1, v2) must be equal

// undirected edges (v1, v2) and (v2, v1) must be equal

@Immutable

public interface Edge<V> {

@NotNull ImmutablePair<Vertex<V>, Vertex<V>> asVerticesPair();

default @NotNull @Immutable Segment asSegment() {

final ImmutablePair<Vertex<V>, Vertex<V>> verticesPair = asVerticesPair();

return new Segment(verticesPair.left.getCoordinates(), verticesPair.right.getCoordinates());

}

default @NotNull Stream<Vertex<V>> asStream() {

final ImmutablePair<Vertex<V>, Vertex<V>> vertices = asVerticesPair();

return Lists.newArrayList(vertices.left, vertices.right).stream();

}

}

// V - vertex ID type

public interface Graph<V> extends Cloneable {

@NotNull @Immutable Set<Vertex<V>> getVertices();

int getVerticesNumber();

@NotNull @Immutable Set<Edge<V>> getEdges();

int getEdgesNumber();

boolean addVertex(@NotNull V id, double x, double y);

boolean addVertex(@NotNull Vertex<V> vertex);

boolean removeVertex(@NotNull V id);

boolean removeVertex(@NotNull Vertex<V> vertex);

@NotNull Vertex<V> vertexOf(@NotNull V id, double x, double y);

boolean addEdge(@NotNull V idFrom, double xFrom, double yFrom, @NotNull V idTo, double xTo, double yTo)

throws GraphException;

boolean addEdge(@NotNull Vertex<V> from, @NotNull Vertex<V> to) throws GraphException;

boolean addEdge(@NotNull Edge<V> edge) throws GraphException;

boolean removeEdge(@NotNull V from, @NotNull V to);

boolean removeEdge(@NotNull Vertex<V> from, @NotNull Vertex<V> to);

boolean removeEdge(@NotNull Edge<V> edge);

boolean removeIntersecting(@NotNull Edge<V> edge);

@NotNull Edge<V> edgeOf(@NotNull V idFrom, double xFrom, double yFrom, @NotNull V idTo, double xTo, double yTo);

@NotNull Edge<V> edgeOf(@NotNull Vertex<V> vertex1, @NotNull Vertex<V> vertex2);

boolean isDirected();

@NotNull @Immutable Collection<Vertex<V>> getNeighbours(@NotNull V id);

@NotNull @Immutable Collection<Vertex<V>> getNeighbours(@NotNull Vertex<V> vertex);

int getIntersectionIndex();

@NotNull Optional<Edge<V>> getMostIntersectingEdge();

boolean isIntersecting();

boolean isConnected();

@NotNull @Immutable List<Graph<V>> getConnectedComponents();

@NotNull Optional<Edge<V>> findBridge();

@NotNull Graph<V> copy();

}

// Vertices with the same ID must be equal

@Immutable

public interface Vertex<V> {

@NotNull @Immutable V getId();

// (x, y)

@NotNull @Immutable Coordinate getCoordinates();

}

class SimpleUndirectedEdge<V> implements Edge<V> {

@NotNull

private final Vertex<V> vertex1;

@NotNull

private final Vertex<V> vertex2;

SimpleUndirectedEdge(final @NotNull Vertex<V> vertex1, final @NotNull Vertex<V> vertex2) {

this.vertex1 = vertex1;

this.vertex2 = vertex2;

}

@Override

public @NotNull ImmutablePair<Vertex<V>, Vertex<V>> asVerticesPair() {

return ImmutablePair.of(vertex1, vertex2);

}

@Override

public boolean equals(Object o) {

if (this == o) {

return true;

}

if (o == null || getClass() != o.getClass()) {

return false;

}

@NotNull SimpleUndirectedEdge<?> anotherEdge = (SimpleUndirectedEdge<?>) o;

return Objects.equals(vertex1, anotherEdge.vertex1) && Objects.equals(vertex2, anotherEdge.vertex2) ||

Objects.equals(vertex1, anotherEdge.vertex2) && Objects.equals(vertex2, anotherEdge.vertex1);

}

@Override

public int hashCode() {

return Objects.hash(vertex1, vertex2) \* Objects.hash(vertex2, vertex1);

}

}

class SimpleVertex<V> implements Vertex<V> {

@NotNull

private final V id;

@NotNull

private final Coordinate coordinates;

SimpleVertex(@NotNull V id, double x, double y) {

this(id, new Coordinate(x, y));

}

SimpleVertex(@NotNull V id, @NotNull Coordinate coordinates) {

this.id = id;

this.coordinates = (Coordinate) coordinates.clone();

}

@Override

public @NotNull @Immutable V getId() {

return id;

}

@Override

public @NotNull @Immutable Coordinate getCoordinates() {

return (Coordinate) coordinates.clone();

}

@Override

public boolean equals(Object o) {

if (this == o) return true;

if (o == null || getClass() != o.getClass()) return false;

SimpleVertex<?> that = (SimpleVertex<?>) o;

return Objects.equals(id, that.id);

}

@Override

public int hashCode() {

return Objects.hash(id);

}

}

public class UndirectedGraphWithIntersections<V> implements Graph<V> {

private final @NotNull Set<Vertex<V>> vertices;

private final @NotNull Set<Edge<V>> edges;

private final @NotNull Multimap<Vertex<V>, Vertex<V>> adjacency;

private final @NotNull Multimap<Edge<V>, Edge<V>> intersections;

public static <V> @NotNull UndirectedGraphWithIntersections<V> of(final @NotNull Collection<Edge<V>> edges) {

final UndirectedGraphWithIntersections<V> graph = new UndirectedGraphWithIntersections<>();

edges.forEach(graph::addEdge);

return graph;

}

public UndirectedGraphWithIntersections() {

vertices = new HashSet<>();

edges = new HashSet<>();

adjacency = HashMultimap.create();

intersections = HashMultimap.create();

}

private UndirectedGraphWithIntersections(

final @NotNull Set<Vertex<V>> vertices,

final @NotNull Set<Edge<V>> edges,

final @NotNull Multimap<Vertex<V>, Vertex<V>> adjacency,

final @NotNull Multimap<Edge<V>, Edge<V>> intersections) {

this.vertices = new HashSet<>(vertices);

this.edges = new HashSet<>(edges);

this.adjacency = HashMultimap.create(adjacency);

this.intersections = HashMultimap.create(intersections);

}

@Override

public @NotNull @Immutable Set<Vertex<V>> getVertices() {

return Collections.unmodifiableSet(vertices);

}

@Override

public int getVerticesNumber() {

return vertices.size();

}

@Override

public @NotNull @Immutable Set<Edge<V>> getEdges() {

return Collections.unmodifiableSet(edges);

}

@Override

public int getEdgesNumber() {

return edges.size();

}

@Override

public boolean addVertex(final @NotNull V id, final double x, final double y) {

return addVertex(vertexOf(id, x, y));

}

@Override

public boolean addVertex(final @NotNull Vertex<V> vertex) throws GraphException {

return vertices.add(vertex);

}

@Override

public boolean removeVertex(final @NotNull V id) {

return removeVertex(vertexOf(id));

}

@Override

public boolean removeVertex(final @NotNull Vertex<V> vertex) {

for (final Vertex<V> neighbour : adjacency.get(vertex)) {

removeEdge(vertex, neighbour);

}

return vertices.remove(vertex);

}

private @NotNull Vertex<V> vertexOf(final @NotNull V id) {

return vertexOf(id, 0, 0);

}

@Override

public @NotNull Vertex<V> vertexOf(final @NotNull V id, final double x, final double y) {

return new SimpleVertex<>(id, x, y);

}

@Override

public boolean addEdge(

final @NotNull V idFrom,

final double xFrom,

final double yFrom,

final @NotNull V idTo,

final double xTo,

final double yTo)

throws GraphException {

return addEdge(edgeOf(idFrom, xFrom, yFrom, idTo, xTo, yTo));

}

@Override

public boolean addEdge(final @NotNull Vertex<V> from, final @NotNull Vertex<V> to) throws GraphException {

return addEdge(edgeOf(from, to));

}

@Override

public boolean addEdge(final @NotNull Edge<V> newEdge) throws GraphException {

if (!edges.contains(newEdge)) {

final @NotNull ImmutablePair<Vertex<V>, Vertex<V>> vertices = newEdge.asVerticesPair();

if (Objects.equals(vertices.left, vertices.right)) {

throw new GraphException("Loops are forbidden");

}

addVertex(vertices.left);

addVertex(vertices.right);

adjacency.put(vertices.left, vertices.right);

adjacency.put(vertices.right, vertices.left);

for (final Edge<V> edge : edges) {

if (Geometry.isIntersecting(edge, newEdge)) {

intersections.put(edge, newEdge);

intersections.put(newEdge, edge);

}

}

return edges.add(newEdge);

} else {

return false;

}

}

@Override

public boolean removeEdge(final @NotNull V from, final @NotNull V to) {

return removeEdge(vertexOf(from), vertexOf(to));

}

@Override

public boolean removeEdge(final @NotNull Vertex<V> from, final @NotNull Vertex<V> to) {

return removeEdge(edgeOf(from, to));

}

@Override

public boolean removeEdge(final @NotNull Edge<V> edge) {

final @NotNull ImmutablePair<Vertex<V>, Vertex<V>> vertices = edge.asVerticesPair();

adjacency.remove(vertices.left, vertices.right);

adjacency.remove(vertices.right, vertices.left);

for (final Edge<V> intersecting : intersections.get(edge)) {

intersections.remove(intersecting, edge);

}

intersections.removeAll(edge);

return edges.remove(edge);

}

@Override

public boolean removeIntersecting(final @NotNull Edge<V> edge) {

final List<Edge<V>> intersectingEdges = new ArrayList<>(intersections.get(edge));

for (final Edge<V> intersectingEdge : intersectingEdges) {

removeEdge(intersectingEdge);

}

return intersectingEdges.size() > 0;

}

@Override

public @NotNull Edge<V> edgeOf(

final @NotNull V idFrom,

final double xFrom,

final double yFrom,

final @NotNull V idTo,

final double xTo,

final double yTo) {

return edgeOf(vertexOf(idFrom, xFrom, yFrom), vertexOf(idTo, xTo, yTo));

}

@Override

public @NotNull Edge<V> edgeOf(final @NotNull Vertex<V> vertex1, final @NotNull Vertex<V> vertex2) {

return new SimpleUndirectedEdge<>(vertex1, vertex2);

}

@Override

public boolean isDirected() {

return false;

}

@Override

public @NotNull @Immutable Collection<Vertex<V>> getNeighbours(final @NotNull V id) {

return getNeighbours(vertexOf(id));

}

@Override

public @NotNull @Immutable Collection<Vertex<V>> getNeighbours(final @NotNull Vertex<V> vertex) {

return Collections.unmodifiableCollection(adjacency.get(vertex));

}

@Override

public @NotNull Optional<Edge<V>> getMostIntersectingEdge() {

Edge<V> mostIntersecting = null;

for (final Edge<V> edge : intersections.keySet()) {

if (mostIntersecting == null ||

intersections.get(mostIntersecting).size() < intersections.get(edge).size()) {

mostIntersecting = edge;

}

}

return Optional.ofNullable(mostIntersecting);

}

@Override

public int getIntersectionIndex() {

return getMostIntersectingEdge()

.map(intersections::get)

.map(Collection::size)

.orElse(0);

}

@Override

public boolean isIntersecting() {

return getIntersectionIndex() > 0;

}

@Override

public boolean isConnected() {

final Set<Vertex<V>> visited = new HashSet<>();

final Queue<Vertex<V>> verticesToVisit = new ArrayDeque<>();

if (!vertices.isEmpty()) {

final Vertex<V> first = vertices.iterator().next();

verticesToVisit.add(first);

while (!verticesToVisit.isEmpty()) {

final Vertex<V> current = verticesToVisit.remove();

if (!visited.contains(current)) {

visited.add(current);

verticesToVisit.addAll(getNeighbours(current));

}

}

return vertices.size() == visited.size();

} else {

return true;

}

}

@Override

public @NotNull @Immutable List<Graph<V>> getConnectedComponents() {

if (!vertices.isEmpty()) {

final Set<Vertex<V>> visited = new HashSet<>();

final List<Graph<V>> connectedComponents = new ArrayList<>();

for (final Vertex<V> vertex : vertices) {

if (!visited.contains(vertex)) {

final Set<Vertex<V>> connectedComponentVertices = new HashSet<>();

final Queue<Vertex<V>> verticesToVisit = new ArrayDeque<>();

verticesToVisit.add(vertex);

while (!verticesToVisit.isEmpty()) {

final Vertex<V> current = verticesToVisit.remove();

if (!visited.contains(current)) {

visited.add(current);

connectedComponentVertices.add(current);

verticesToVisit.addAll(getNeighbours(current));

}

}

connectedComponents.add(subGraph(connectedComponentVertices));

}

}

return Collections.unmodifiableList(connectedComponents);

} else {

return Collections.singletonList(new UndirectedGraphWithIntersections<>());

}

}

private @NotNull UndirectedGraphWithIntersections<V> subGraph(final @NotNull Set<Vertex<V>> vertices) {

final UndirectedGraphWithIntersections<V> subGraph = new UndirectedGraphWithIntersections<>();

vertices.forEach(subGraph::addVertex);

for (final Vertex<V> vertex : vertices) {

for (final Vertex<V> neighbour : getNeighbours(vertex)) {

if (vertices.contains(neighbour)) {

subGraph.addEdge(edgeOf(vertex, neighbour));

}

}

}

return subGraph;

}

@Override

public @NotNull Optional<Edge<V>> findBridge() {

if (!vertices.isEmpty()) {

return bridgeDfs(

new MutableInt(0),

vertices.iterator().next(),

null,

new HashSet<>(),

new HashMap<>(),

new HashMap<>());

} else {

return Optional.empty();

}

}

private @NotNull Optional<Edge<V>> bridgeDfs(

final @NotNull MutableInt time,

final @NotNull Vertex<V> visiting,

final @Nullable Vertex<V> parent,

final @NotNull Set<Vertex<V>> visited,

final @NotNull Map<Vertex<V>, Integer> inTime,

final @NotNull Map<Vertex<V>, Integer> upTime) {

visited.add(visiting);

time.increment();

inTime.put(visiting, time.getValue());

upTime.put(visiting, time.getValue());

for (final Vertex<V> child : getNeighbours(visiting)) {

if (!Objects.equals(child, parent)) {

if (visited.contains(child)) {

upTime.put(visiting, Math.min(upTime.get(visiting), inTime.get(child)));

} else {

final Optional<Edge<V>> bridge = bridgeDfs(time, child, visiting, visited, inTime, upTime);

if (bridge.isPresent()) {

return bridge;

} else {

upTime.put(visiting, Math.min(upTime.get(visiting), upTime.get(child)));

if (upTime.get(child) > inTime.get(visiting)) {

return Optional.of(edgeOf(visiting, child));

}

}

}

}

}

return Optional.empty();

}

@Override

public @NotNull Graph<V> copy() {

return new UndirectedGraphWithIntersections<>(vertices, edges, adjacency, intersections);

}

}

@Immutable

public interface Matroid<E> {

@NotNull @Immutable Set<E> getElements();

Optional<@NotEmpty Set<E>> findCircuit(@NotNull Set<E> subset) throws MatroidException;

}

abstract class GraphMatroid<V> implements Matroid<Edge<V>> {

@NotNull

@Immutable

protected final Graph<V> graph;

protected GraphMatroid(final @NotNull Graph<V> graph) {

this.graph = graph.copy();

}

@Override

public @NotNull @Immutable Set<Edge<V>> getElements() {

return graph.getEdges();

}

@Override

public final Optional<@NotEmpty Set<Edge<V>>> findCircuit(final @NotNull Set<Edge<V>> subset)

throws MatroidException {

if (!getElements().containsAll(subset)) {

throw new MatroidException("The given set isn't a subset of matroid elements");

}

if (subset.isEmpty()) {

return Optional.empty();

} else {

return findCircuitChecked(subset);

}

}

protected abstract Optional<@NotEmpty Set<Edge<V>>> findCircuitChecked(@NotNull @NotEmpty Set<Edge<V>> subset)

throws MatroidException;

}

class CycleMatroid<V> extends GraphMatroid<V> {

CycleMatroid(@NotNull Graph<V> graph) throws MatroidException {

super(graph);

if (graph.isDirected()) {

throw new MatroidException("Directed graphs aren't supported");

}

}

@Override

protected Optional<@NotEmpty Set<Edge<V>>> findCircuitChecked(final @NotNull @NotEmpty Set<Edge<V>> subset) {

final ImmutableSet<Vertex<V>> vertices = subset.stream()

.flatMap(Edge::asStream)

.collect(ImmutableSet.toImmutableSet());

return findCircuit(vertices, subset);

}

private Optional<@NotEmpty Set<Edge<V>>> findCircuit(

final @NotNull Set<Vertex<V>> vertices,

final @NotNull Set<Edge<V>> edges) {

final Map<Vertex<V>, Vertex<V>> ancestors = new HashMap<>();

Optional<Vertex<V>> repeated = Optional.empty();

for (final Vertex<V> vertex : vertices) {

if (!ancestors.containsKey(vertex)) {

ancestors.put(vertex, null);

repeated = dfs(vertex, edges, ancestors);

if (repeated.isPresent()) {

break;

}

}

}

if (repeated.isPresent()) {

final Set<Edge<V>> circuit = new HashSet<>();

final Vertex<V> first = repeated.get();

Vertex<V> current = first;

do {

assert ancestors.containsKey(current);

final @NotNull Vertex<V> next = ancestors.get(current);

circuit.add(graph.edgeOf(next, current));

current = next;

} while (!Objects.equals(current, first));

return Optional.of(circuit);

} else {

return Optional.empty();

}

}

// returns first repeated vertex

// ancestors must form a cycle if a repeated vertex is found

private Optional<Vertex<V>> dfs(

final @NotNull Vertex<V> current,

final @NotNull Set<Edge<V>> edges,

final @NotNull @Mutable Map<Vertex<V>, Vertex<V>> ancestors) {

assert ancestors.containsKey(current);

for (final Vertex<V> neighbour : graph.getNeighbours(current)) {

if (edges.contains(graph.edgeOf(current, neighbour))) {

if (ancestors.containsKey(neighbour)) {

if (!Objects.equals(neighbour, ancestors.get(current))) {

ancestors.put(neighbour, current);

return Optional.of(neighbour);

}

} else {

ancestors.put(neighbour, current);

final Optional<Vertex<V>> repeated = dfs(neighbour, edges, ancestors);

if (repeated.isPresent()) {

return repeated;

}

}

}

}

return Optional.empty();

}

}

class IntersectionMatroid<V> extends GraphMatroid<V> {

// edges that aren't in the map don't intersect

@NotNull

@Immutable

private final Map<Edge<V>, Integer> intersectionGroups;

IntersectionMatroid(final @NotNull Graph<V> graph, final boolean validate) throws MatroidException {

super(graph);

intersectionGroups = Collections.unmodifiableMap(buildIntersectionGroups(validate));

}

private @NotNull Map<Edge<V>, Integer> buildIntersectionGroups(final boolean validate) throws MatroidException {

final Map<Edge<V>, Integer> intersectionGroups = new HashMap<>();

final List<Edge<V>> ordered = new ArrayList<>(graph.getEdges());

int intersectionGroup = 0;

for (int i = 0; i < ordered.size(); i++) {

final Edge<V> edge1 = ordered.get(i);

for (int j = 0; j < i; j++) {

final Edge<V> edge2 = ordered.get(j);

if (Geometry.isIntersecting(edge1, edge2)) {

final int currentIntersectionGroup;

if (!intersectionGroups.containsKey(edge2)) {

currentIntersectionGroup = intersectionGroup++;

intersectionGroups.put(edge2, currentIntersectionGroup);

} else {

currentIntersectionGroup = intersectionGroups.get(edge2);

}

if (!intersectionGroups.containsKey(edge1)) {

intersectionGroups.put(edge1, currentIntersectionGroup);

if (!validate) {

break;

}

} else if (currentIntersectionGroup != intersectionGroups.get(edge1)) {

throw new MatroidException("Intersection matroid can't be built for this graph");

}

}

}

}

return intersectionGroups;

}

@SuppressWarnings("unchecked")

@Override

protected Optional<@NotEmpty Set<Edge<V>>> findCircuitChecked(final @NotNull @NotEmpty Set<Edge<V>> subset) {

final Map<Integer, Edge<V>> intersectingVisited = new HashMap<>();

for (final Edge<V> edge : subset) {

if (intersectionGroups.containsKey(edge)) {

final int intersectionGroup = intersectionGroups.get(edge);

if (intersectingVisited.containsKey(intersectionGroup)) {

final Edge<V> intersecting = intersectingVisited.get(intersectionGroup);

final Set<Edge<V>> circuit = Sets.newHashSet(intersecting, edge);

return Optional.of(circuit);

} else {

intersectingVisited.put(intersectionGroup, edge);

}

}

}

return Optional.empty();

}

}

class MatroidWithFixedElements<E> implements Matroid<E> {

@NotNull

private final Matroid<E> matroid;

@NotNull

@Immutable

private final Set<E> fixedElements;

@NotNull

@Immutable

private final Set<E> notFixedElements;

MatroidWithFixedElements(final @NotNull Matroid<E> matroid, final @NotNull Set<E> fixedElements)

throws MatroidException {

if (!matroid.getElements().containsAll(fixedElements)) {

throw new MatroidException("Matroid doesn't contain all fixed elements");

}

if (matroid.findCircuit(fixedElements).isPresent()) {

throw new MatroidException("Fixed elements aren't independent subset of elements");

}

this.matroid = matroid;

this.fixedElements = ImmutableSet.copyOf(fixedElements);

this.notFixedElements = CollectionUtils.immutableDifferenceOf(matroid.getElements(), fixedElements);

}

@Override

public @NotNull @Immutable Set<E> getElements() {

return notFixedElements;

}

@Override

public Optional<@NotEmpty Set<E>> findCircuit(@NotNull Set<E> subset) throws MatroidException {

final Set<E> subsetWithFixedElements = new HashSet<>(subset);

subsetWithFixedElements.addAll(fixedElements);

return matroid.findCircuit(subsetWithFixedElements);

}

}

public class Matroids {

public static <V> @NotNull Matroid<Edge<V>> cycleMatroidOf(final @NotNull Graph<V> graph) {

return new CycleMatroid<>(graph);

}

public static <V> @NotNull Matroid<Edge<V>> intersectionMatroidOf(final @NotNull Graph<V> graph)

throws MatroidException {

return new IntersectionMatroid<>(graph, true);

}

public static <V> @NotNull Matroid<Edge<V>> cycleMatroidWithFixedEdgesOf(

final @NotNull Graph<V> graph,

final @NotNull Set<Edge<V>> fixedEdges) {

return new MatroidWithFixedElements<>(cycleMatroidOf(graph), fixedEdges);

}

public static <V> @NotNull Matroid<Edge<V>> intersectionMatroidWithFixedEdgesOf(

final @NotNull Graph<V> graph,

final @NotNull Set<Edge<V>> fixedEdges) {

return new MatroidWithFixedElements<>(intersectionMatroidOf(graph), fixedEdges);

}

}

@Immutable

public interface MatroidIntersectionAlgorithm<E> {

@NotNull Set<E> findIntersection(final @NotNull Matroid<E> matroid1, final @NotNull Matroid<E> matroid2)

throws AlgorithmException;

}

public class BaseMatroidIntersectionAlgorithm<E> implements MatroidIntersectionAlgorithm<E> {

@SuppressWarnings("StatementWithEmptyBody")

public @NotNull Set<E> findIntersection(final @NotNull Matroid<E> matroid1, final @NotNull Matroid<E> matroid2)

throws AlgorithmException {

if (!SetUtils.isEqualSet(matroid1.getElements(), matroid2.getElements())) {

throw new AlgorithmException("Matroids have different elements");

}

final Set<E> intersection = new HashSet<>();

try {

while (expand(matroid1, matroid2, intersection));

} catch (Exception e) {

throw new AlgorithmException("Internal error", e);

}

return intersection;

}

private boolean expand(

final @NotNull Matroid<E> matroid1,

final @NotNull Matroid<E> matroid2,

final @NotNull @Mutable Set<E> intersection) {

assert SetUtils.isEqualSet(matroid1.getElements(), matroid2.getElements());

// parts and adjacency list of the utility bipartite graph

final Queue<E> vertices1 = new ArrayDeque<>();

final Set<E> vertices2 = new HashSet<>();

final Multimap<E, E> adjacency = HashMultimap.create();

// ancestors to restore the expanding path

// null value is used to represent the start point of a path

final Map<E, @Nullable E> ancestors = new HashMap<>();

final @NotNull @Immutable Set<E> elements = matroid1.getElements();

final @NotNull @Immutable Set<E> iteratingElements =

CollectionUtils.immutableDifferenceOf(elements, intersection);

for (final E candidate : iteratingElements) {

intersection.add(candidate);

final Optional<@NotEmpty Set<E>> circuit1 = matroid1.findCircuit(intersection);

final Optional<@NotEmpty Set<E>> circuit2 = matroid2.findCircuit(intersection);

final boolean independentInMatroid1 = !circuit1.isPresent();

final boolean independentInMatroid2 = !circuit2.isPresent();

if (independentInMatroid1 && independentInMatroid2) {

return true;

}

if (independentInMatroid1) {

vertices1.add(candidate);

ancestors.put(candidate, null);

} else {

for (final E element : circuit1.get()) {

if (!Objects.equals(element, candidate)) {

adjacency.put(element, candidate);

}

}

}

if (independentInMatroid2) {

vertices2.add(candidate);

} else {

for (final E element : circuit2.get()) {

if (!Objects.equals(element, candidate)) {

adjacency.put(candidate, element);

}

}

}

intersection.remove(candidate);

}

while (!vertices1.isEmpty()) {

final E visiting = vertices1.remove();

for (final E neighbour : adjacency.get(visiting)) {

if (!ancestors.containsKey(neighbour)) {

ancestors.put(neighbour, visiting);

vertices1.add(neighbour);

if (vertices2.contains(neighbour)) {

expand(intersection, neighbour, ancestors);

return true;

}

}

}

}

return false;

}

private void expand(

final @NotNull @Mutable Set<E> intersection,

final @NotNull E lastVertex,

final @NotNull Map<E, E> ancestors) {

final int initialIntersectionSize = intersection.size();

final List<E> path = new ArrayList<>();

E current = lastVertex;

while (current != null) {

path.add(current);

current = ancestors.get(current);

}

assert (path.size() & 1) == 1;

for (int i = path.size() - 1; i >= 0; --i) {

final E processing = path.get(i);

if ((i & 1) == 0) {

final boolean added = intersection.add(processing);

assert added;

} else {

final boolean removed = intersection.remove(processing);

assert removed;

}

}

assert intersection.size() > initialIntersectionSize;

}

}

@Immutable

public interface NcstAlgorithm<V> {

Optional<Graph<V>> findNcst(final @NotNull Graph<V> graph) throws AlgorithmException;

}

class IndependentSet<V> {

private final @NotNull Map<Vertex<V>, Vertex<V>> verticesGroups;

private final @NotNull Set<Edge<V>> edges;

public IndependentSet() {

verticesGroups = new HashMap<>();

edges = new HashSet<>();

}

private IndependentSet(final @NotNull Map<Vertex<V>, Vertex<V>> verticesGroups, final @NotNull Set<Edge<V>> edges) {

this.verticesGroups = verticesGroups;

this.edges = edges;

}

public @NotNull Graph<V> toGraph() {

return UndirectedGraphWithIntersections.of(edges);

}

public @NotNull @Immutable Set<Edge<V>> getEdges() {

return ImmutableSet.copyOf(edges);

}

@Fluent

public IndependentSet<V> addAll(final @NotNull Collection<Edge<V>> edges) {

for (final Edge<V> edge : edges) {

if (canBeAdded(edge)) {

add(edge);

}

}

return this;

}

@Fluent

public IndependentSet<V> add(final @NotNull Edge<V> edge) throws AlgorithmException {

if (canBeAdded(edge)) {

edges.add(edge);

final ImmutablePair<Vertex<V>, Vertex<V>> vertices = edge.asVerticesPair();

verticesGroups.put(getGroup(vertices.left), getGroup(vertices.right));

return this;

} else {

throw new AlgorithmException("Edge connects vertices from the same group");

}

}

public boolean canBeAdded(final @NotNull Edge<V> edge) {

final ImmutablePair<Vertex<V>, Vertex<V>> vertices = edge.asVerticesPair();

return !Objects.equals(getGroup(vertices.left), getGroup(vertices.right));

}

private @NotNull Vertex<V> getGroup(final @NotNull Vertex<V> vertex) {

if (verticesGroups.containsKey(vertex)) {

final Vertex<V> parent = verticesGroups.get(vertex);

if (Objects.equals(vertex, parent)) {

return vertex;

} else {

final Vertex<V> group = getGroup(parent);

verticesGroups.put(vertex, group);

return group;

}

} else {

verticesGroups.put(vertex, vertex);

return vertex;

}

}

public @NotNull IndependentSet<V> copy() {

return new IndependentSet<>(new HashMap<>(verticesGroups), new HashSet<>(edges));

}

public @NotNull IndependentSet<V> filter(final @NotNull Collection<Vertex<V>> vertices) {

final IndependentSet<V> filtered = new IndependentSet<>();

edges.stream()

.filter(edge -> vertices.containsAll(edge.asStream().collect(Collectors.toList())))

.forEach(filtered::add);

return filtered;

}

}

public class NcstAlgorithmImpl<V> implements NcstAlgorithm<V> {

private static final boolean LOG = false;

@Override

@SuppressWarnings("OptionalAssignedToNull")

public @NotNull Optional<Graph<V>> findNcst(final @NotNull Graph<V> graph) throws AlgorithmException {

final ForkJoinPool pool = new ForkJoinPool();

final AtomicInteger nodesCreated = new AtomicInteger(0);

final AtomicInteger nodesProcessed = new AtomicInteger(0);

final ForkJoinTask<Optional<Graph<V>>> executionTask =

pool.submit(new SearchTreeNode(nodesCreated, nodesProcessed, graph, new IndependentSet<>()));

Optional<Graph<V>> ncst = null;

do {

try {

ncst = executionTask.get(1, TimeUnit.SECONDS);

} catch (final TimeoutException e) {

final int processed = nodesProcessed.get();

final int created = nodesCreated.get();

System.out.println(String.format(

"%d created / %d processed / %d left", created, processed, created - processed));

} catch (final Exception e) {

throw new AlgorithmException(e);

}

} while (ncst == null);

pool.shutdown();

return ncst;

}

private class SearchTreeNode extends RecursiveTask<Optional<Graph<V>>> {

private final @NotNull AtomicInteger nodesCreated;

private final @NotNull AtomicInteger nodesProcessed;

private final int depth;

private final @NotNull String index;

private final @NotNull Graph<V> graph;

private final @NotNull IndependentSet<V> fixed;

private SearchTreeNode(

final @NotNull AtomicInteger nodesCreated,

final @NotNull AtomicInteger nodesProcessed,

final @NotNull Graph<V> graph,

final @NotNull IndependentSet<V> fixed) {

this(nodesCreated, nodesProcessed, 0, "f", graph, fixed);

}

private SearchTreeNode(

final @NotNull AtomicInteger nodesCreated,

final @NotNull AtomicInteger nodesProcessed,

final int depth,

final @NotNull String index,

final @NotNull Graph<V> graph,

final @NotNull IndependentSet<V> fixed) {

this.nodesCreated = nodesCreated;

this.nodesProcessed = nodesProcessed;

this.depth = depth;

this.index = index;

this.graph = graph;

this.fixed = fixed;

nodesCreated.incrementAndGet();

}

@Override

protected Optional<Graph<V>> compute() {

final Optional<Graph<V>> result = \_compute();

nodesProcessed.incrementAndGet();

return result;

}

@SuppressWarnings("OptionalGetWithoutIsPresent")

private Optional<Graph<V>> \_compute() {

if (graph.getVerticesNumber() <= 1) {

log(true, String.format("vertices number = %d", graph.getVerticesNumber()));

return Optional.of(graph.copy());

} else if (!graph.isConnected()) {

log(true, "not connected");

return Optional.empty();

} else if (!graph.isIntersecting()) {

log(true, "non-crossing");

fixed.addAll(graph.getEdges());

return Optional.of(fixed.toGraph());

} else if (graph.getIntersectionIndex() == 1) {

log(true, "intersection index 1");

final Matroid<Edge<V>> cycleMatroid = Matroids.cycleMatroidWithFixedEdgesOf(graph, fixed.getEdges());

final Matroid<Edge<V>> intersectionMatroid =

Matroids.intersectionMatroidWithFixedEdgesOf(graph, fixed.getEdges());

final MatroidIntersectionAlgorithm<Edge<V>> matroidIntersectionAlgorithm =

new BaseMatroidIntersectionAlgorithm<>();

final Set<Edge<V>> nonCrossingEdges =

matroidIntersectionAlgorithm.findIntersection(cycleMatroid, intersectionMatroid);

if (nonCrossingEdges.size() == graph.getVerticesNumber() - 1) {

return Optional.of(UndirectedGraphWithIntersections.of(nonCrossingEdges));

} else {

return Optional.empty();

}

} else {

final Optional<Edge<V>> optionalBridge = graph.findBridge();

if (optionalBridge.isPresent()) {

final Edge<V> bridge = optionalBridge.get();

final Graph<V> graphCopy = graph.copy();

graphCopy.removeIntersecting(bridge);

final boolean removed = graphCopy.removeEdge(bridge);

assert removed;

final List<Graph<V>> connectedComponents = graphCopy.getConnectedComponents();

if (connectedComponents.size() == 2) {

log(false, "splitting by bridge");

final List<SearchTreeNode> tasks = ImmutableList.of(

new SearchTreeNode(

nodesCreated, nodesProcessed,

depth + 1, index + "o",

connectedComponents.get(0),

fixed.filter(connectedComponents.get(0).getVertices())),

new SearchTreeNode(

nodesCreated, nodesProcessed,

depth + 1, index + "t",

connectedComponents.get(1),

fixed.filter(connectedComponents.get(1).getVertices())));

final List<Graph<V>> subTrees = invokeAll(tasks).stream()

.map(SearchTreeNode::join)

.filter(Optional::isPresent)

.map(Optional::get)

.collect(Collectors.toList());

if (subTrees.size() == 2) {

final Graph<V> tree = UndirectedGraphWithIntersections.of(Collections.singleton(bridge));

for (final Graph<V> subTree : subTrees) {

subTree.getEdges()

.forEach(tree::addEdge);

}

return Optional.of(tree);

} else {

return Optional.empty();

}

} else if (connectedComponents.size() > 2) {

log(true, "splitting by bridge is impossible: too many connected components");

return Optional.empty();

} else {

throw new RuntimeException();

}

} else {

final List<SearchTreeNode> tasks = new ArrayList<>();

final Edge<V> mostIntersecting = graph.getMostIntersectingEdge().get();

Graph<V> graphCopy = graph.copy();

graphCopy.removeEdge(mostIntersecting);

IndependentSet<V> fixedCopy = fixed.copy();

tasks.add(new SearchTreeNode(

nodesCreated, nodesProcessed, depth + 1, index + "e", graphCopy, fixedCopy));

if (fixed.canBeAdded(mostIntersecting)) {

log(false, "splitting by most intersecting edge, most intersecting edge is included");

graphCopy = graph.copy();

graphCopy.removeIntersecting(mostIntersecting);

fixedCopy = fixed.copy();

fixedCopy.add(mostIntersecting);

tasks.add(new SearchTreeNode(

nodesCreated, nodesProcessed, depth + 1, index + "i", graphCopy, fixedCopy));

} else {

log(false, "splitting by most intersecting edge, most intersecting edge isn't included");

}

return invokeAll(tasks).stream()

.map(SearchTreeNode::join)

.filter(Optional::isPresent)

.findAny()

.orElse(Optional.empty());

}

}

}

private void log(final boolean finalNode, final @NotNull String message) {

final String color = finalNode ? "\u001B[33m" : "\u001B[34m";

final String reset = "\u001B[0m";

if (LOG) {

System.out.println(String.format("%s%d:%s / %s%s", color, depth, index, message, reset));

}

}

}

}

public class AlgorithmException extends RuntimeException {

public AlgorithmException() {

}

public AlgorithmException(String message) {

super(message);

}

public AlgorithmException(String message, Throwable cause) {

super(message, cause);

}

public AlgorithmException(Throwable cause) {

super(cause);

}

}

public class GraphException extends RuntimeException {

public GraphException() {

}

public GraphException(String message) {

super(message);

}

public GraphException(String message, Throwable cause) {

super(message, cause);

}

public GraphException(Throwable cause) {

super(cause);

}

}

public class MatroidException extends RuntimeException {

public MatroidException() {

}

public MatroidException(String message) {

super(message);

}

public MatroidException(String message, Throwable cause) {

super(message, cause);

}

public MatroidException(Throwable cause) {

super(cause);

}

}

@Documented

@Retention(RetentionPolicy.SOURCE)

@Target(ElementType.METHOD)

public @interface Fluent {

}

@Documented

@Inherited

@Retention(RetentionPolicy.SOURCE)

@Target({ElementType.LOCAL\_VARIABLE, ElementType.PARAMETER, ElementType.METHOD, ElementType.FIELD, ElementType.TYPE\_USE, ElementType.TYPE})

public @interface Immutable {

}

@Documented

@Inherited

@Retention(RetentionPolicy.SOURCE)

@Target({ElementType.LOCAL\_VARIABLE, ElementType.PARAMETER, ElementType.METHOD, ElementType.FIELD, ElementType.TYPE\_USE})

public @interface Mutable {

}

@Documented

@Inherited

@Retention(RetentionPolicy.SOURCE)

@Target({ElementType.LOCAL\_VARIABLE, ElementType.PARAMETER, ElementType.METHOD, ElementType.FIELD, ElementType.TYPE\_USE})

public @interface NotEmpty {

}

public class CollectionUtils {

private CollectionUtils() {}

public static <E> @NotNull Set<E> differenceOf(final @NotNull Set<E> minuend, final @NotNull Set<E> subtrahend) {

final Set<E> result = new HashSet<>(minuend);

result.removeAll(subtrahend);

return result;

}

public static <E> @NotNull @Immutable Set<E> immutableDifferenceOf(

final @NotNull Set<E> minuend, final @NotNull Set<E> subtrahend) {

return Collections.unmodifiableSet(differenceOf(minuend, subtrahend));

}

}

public class Geometry {

private Geometry() {}

public static <V> boolean isIntersecting(@NotNull Edge<V> edge1, @NotNull Edge<V> edge2) {

final @NotNull @Immutable Segment segment1 = edge1.asSegment();

final @NotNull @Immutable Segment segment2 = edge2.asSegment();

final @Immutable Coordinate intersection = segment1.intersection(segment2);

return intersection != null &&

!intersection.equals2D(segment1.getStart()) &&

!intersection.equals2D(segment1.getEnd()) &&

!intersection.equals2D(segment2.getStart()) &&

!intersection.equals2D(segment2.getEnd());

}

}