МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ Федеральное государственное автономное образовательное учреждение

высшего образования   
**«Национальный исследовательский   
Нижегородский государственный университет им. Н.И. Лобачевского»**

**(ННГУ)**

**Институт информационных технологий, математики и механики**

**Кафедра: алгебры, геометрии и дискретной математики**

Направление подготовки: «Программная инженерия»

Профиль подготовки: «Разработка программно-информационных систем»

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА**

на тему:

**«Дистанционное римское доминирование в графах без ограничения веса»**

**Выполнил:** студент группы 381908-1

Ермолин Дмитрий Алексеевич

Подпись

**Научный руководитель:** доцент кафедры алгебры, геометрии и дискретной математики

Мокеев Дмитрий Борисович

Подпись

Нижний Новгород  
2023

# Оглавление

[Оглавление 2](#_Toc137668183)

[Введение 3](#_Toc137668184)

[Глава 1. Классическая задача римского доминирования 5](#_Toc137668185)

[**1.1** **Формулировка задачи римского доминирования** 5](#_Toc137668186)

[**1.2** **Сложность задачи римского доминирования** 5](#_Toc137668187)

[**1.3** **Решения задачи римского доминирования** 6](#_Toc137668188)

[Глава 2. Модификация задачи римского доминирования 8](#_Toc137668189)

[**2.1** **Постановка задачи** 8](#_Toc137668191)

[**2.2** **Решение дистанционной задачи римского доминирования** 8](#_Toc137668192)

[**2.3** **Поиск альтернативных решений** 10](#_Toc137668193)

[**2.4** **Оптимизация алгоритма решения** 12](#_Toc137668194)

[**2.5** **Эксперименты с полученными алгоритмами** 13](#_Toc137668195)

[Заключение 16](#_Toc137668196)

[Список литературы 17](#_Toc137668197)

[Исходный код 18](#_Toc137668198)

# Введение

Римское доминирование — это задача, имеющая свои истоки в античной Римской империи. Задача возникла из необходимости обеспечить безопасность и контроль над городами и поселениями империи.

История задачи римского доминирования восходит к временам, когда Римская империя стремительно расширялась, захватывая новые территории и ассимилируя различные культуры и общества. В этом процессе Риму было важно обеспечить контроль над завоеванными территориями и предотвратить мятежи или восстания.

Одним из способов обеспечения контроля было размещение римских солдат или стражей в стратегически важных точках городов и поселений. Цель состояла в том, чтобы выбрать наименьшее число стражей, которые могли бы контролировать все участки города или поселения.

Эта проблема заинтересовала исследователей и математиков, которые начали исследовать оптимальные стратегии расстановки стражей для обеспечения римского доминирования. Идея заключалась в том, чтобы найти такой минимальный набор вершин графа (соответствующий местоположению стражей), чтобы каждая вершина была либо сама являлась стражем, либо имела соседнюю вершину, которая была бы охраняема стражем.

Со временем задача римского доминирования превратилась в активную область исследований в дискретной математике и теории графов. Математики и компьютерные ученые разработали различные алгоритмы и подходы для эффективного решения этой задачи, исследовали ее связь с другими комбинаторными проблемами и расширили ее применение в различных областях, включая сетевую безопасность, транспортные системы и социальные сети.

Целью данной работы является исследование и анализ модификации задачи римского доминирования с неограниченным количеством стражей. В ней рассматриваются различные подходы и алгоритмы, которые позволят эффективно определить оптимальное распределение стражей для обеспечения полного контроля над графом.

В первой главе мы рассмотрим основные понятия и определения, связанные с задачей римского доминирования, такими как римский доминирующий набор, римская доминирующая функция и другими ключевыми понятиями, необходимыми для понимания.

Во второй главе мы рассмотрим решение задачи римского доминирования с неограниченным количеством стражей. В рамках работы будут проведены эксперименты с использованием различных графовых структур для оценки эффективности различных подходов. Будут анализироваться полученные результаты, чтобы выявить преимущества и недостатки каждого подхода и определить оптимальные стратегии для решения задачи римского доминирования с неограниченным количеством стражей.

# Глава 1. Классическая задача римского доминирования

В этой главе рассматривается классическая задача римского доминирования, ее решение и свойства.

## **Формулировка задачи римского доминирования**

Впервые задача римского доминирования была предложена в статье Иэна Стюарта в Scientific American под названием "Defend the Roman Empire!"[1]. Формулировка задачи, используемая в данной работе, была сформулирована Кокйеном и др.[2].

Для графа **G = (V, E)** пусть **f: V → {0, 1, 2}**, а **(V0, V1, V2)** - упорядоченное разбиение **V**, порожденное **f**, где **Vi = {υ ∈ V | f(υ) = i}**, а **|Vi| = ni**, для **i = 0, 1, 2**. Заметим, что существует взаимно-однозначное соответствие между функциями **f: V → {0, 1, 2}** и упорядоченными разбиениями **(V0, V1, V2)** множества **V**. Поэтому мы будем писать **f = (V0, V1, V2)**.

Функция **f = (V0, V1, V2)** является римской доминирующей функцией (RDF), если **V2** доминирует над **V0**, то есть **V0 ⊆ N[V2]**. Вес функции **f** равен **f(V) = ∑ υ ∈V f(υ) = 2n2 + n1**.

Римское число доминации, обозначаемое **γR (G)**, равно минимальному весу **RDF** в **G**, и мы говорим, что функция является **γR** -функцией, если она является RDF и

## **Сложность задачи римского доминирования**

Задача римского доминирования, также как и задача о доминирующем множестве, относится к классу NP-полных задач, однако имеются некоторые уточнения.

Так, авторы статьи [2] разработали линейный алгоритм для вычисления числа римского доминирования для любого дерева.

Также упоминается, что проблема принятия решения **RDF**, связанная со значением римской доминирующей функции, является NP-полной, даже при ограничениях на хордальные, двудольные, разделенные или планарные графы.

Было выдвинуто предположение, что неравенства немедленно подразумевают наличие аппроксимационного алгоритма с точностью 2log n для числа римского доминирования, в то время как аппроксимационный алгоритм с точностью clog n не существует для любого , если **P = NP**.

## **Решения задачи римского доминирования**

Различные решения для различного рода графов были предложены в статьях [4],[5],[6]. В статье [3] используется теория игр для изучения проблемы римского доминирования на общих графах.

В контексте задачи римского доминирования, где требуется разместить минимальное количество войск на вершинах графа, так чтобы каждая вершина без войск была смежна с вершиной, на которой находится хотя бы два войска, жадный алгоритм может быть эффективным методом приближенного решения.

Идея жадного алгоритма заключается в следующем: начиная с пустого множества доминирующих вершин, на каждом шаге выбирается вершина, которая наиболее эффективно увеличивает число доминирующих вершин. В данном случае, выбирается вершина с наибольшим числом смежных недоминированных вершин и помещается в множество доминирующих вершин.

Процесс продолжается до тех пор, пока все вершины графа не будут либо доминированы, либо включены в множество доминирующих вершин. В результате работы жадного алгоритма получается приближенное решение задачи римского доминирования, которое обычно является достаточно хорошим приближением оптимального решения.

Однако следует отметить, что жадный алгоритм не гарантирует нахождение оптимального решения задачи римского доминирования. В некоторых случаях он может давать недостаточно эффективное решение. Поэтому для точного решения задачи требуется применение других методов и алгоритмов, таких как динамическое программирование или алгоритмы с ветвями и границами.

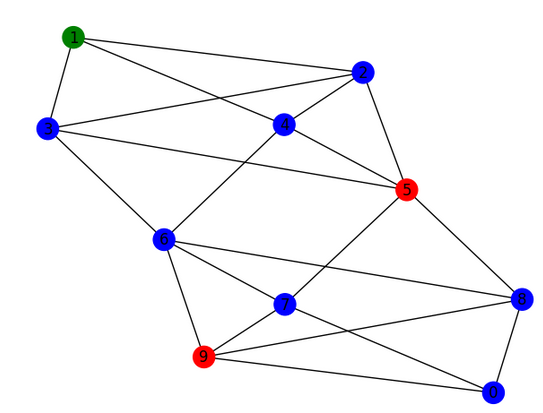


Рисунок 1. Решение задачи римского доминирования жадным алгоритмом

Жадный алгоритм имеет свои преимущества, такие как простота реализации и относительная эффективность на практике. В многих случаях он может быть полезным приближенным методом для решения задачи римского доминирования, особенно в больших графах, где точное решение становится вычислительно сложным.

Приведем пример такого алгоритма (Алгоритм №1)

1. Инициализируем граф и список цветов для каждой вершины. Изначально все вершины помечены черным.
2. Пока в списке цветов присутствует черный (т.е. остаются непокрытые вершины), выполняем следующие действия:
   * Находим вершину с наибольшим числом непокрытых соседей. Для этого проходим по каждой вершине графа и выбираем вершину, у которой количество непокрытых соседей наибольшее.
   * Если найдена вершина с непокрытыми соседями, помечаем ее красным и помечаем синим всех ее соседей, которые до этого были непокрытыми.
   * Если у найденной вершины нет непокрытых соседей, помечаем ее зеленым, что означает покрытие только себя.

Алгоритм реализован на языке Python с использованием библиотеки для работы с графами Networkx.

# Глава 2. Модификация задачи римского доминирования

В этой главе мы рассмотрим модификацию задачи римского доминирования и пути ее решения.



## **Постановка задачи**

Модификация задачи римского доминирования без ограничения на размещение стражей в одной вершине предполагает, что в каждой вершине графа может находиться любое количество стражей. Такая модификация учитывает более гибкие условия размещения стражей и позволяет эффективнее использовать ресурсы.

В этой модификации задачи, вместо одного или двух стражей в вершине может располагаться неограниченное количество стражей и при числе стражей **n** все вершины на расстоянии считаются покрытыми. Таким образом, задача сводится к нахождению минимального доминирующего множества вершин.

Модификация задачи римского доминирования без ограничения на размещение стражей в одной вершине предоставляет больше свободы и гибкости в выборе местоположения стражей, что может быть полезным в различных практических сценариях, связанных с обеспечением безопасности, оптимизацией ресурсов и другими приложениями.

Сформулируем задачу формальным языком:

Дистанционная римская доминирующая функция (dRDF) на графе **G = (V; E)** - это функция0, удовлетворяющая условию, что каждая вершина **u**, для которой , находится на расстоянии, не превышающем как минимум с одной вершиной **v**, для которой . Значение римской доминирующей функции - это значение . Минимальное значение **dRDF** на графе **G** называется дистанционным числом римского доминирования **G** или **γRd(G)**.

## **Решение дистанционной задачи римского доминирования**

Одним из путей решения задачи **mRDF** является расположение всех стражей в одной вершине графа – его центре. Экспериментально доказано, что в данной модификации задачи не существует решений, при количестве стражей меньшем , тогда можно предположить, что это число является решением.

Докажем это утверждение:

Пусть существует путь **P** длины **n**, разобьем этот граф на подграфы длины не более 3 так, чтобы количество графов было минимальным.

Тогда для графов

Отсюда следует:

т.к. является количеством ребер, увеличенным на единицу, то он покрывает центр и все вершины в радиусе.

Теперь если мы рассмотрим любой граф, то в нем будет путь диаметр, проходящий через центр, при покрытии которого, не считая остального графа , но при этом при граф будет покрыт уже полностью.

Проведя такое доказательство для всех мы докажем оптимальность расположения в центре.

Конец доказательства.

Исходя из этого утверждения, задача сводится к поиску центра и радиуса графа и расположения в нем стражей. Сложность такого решения можно оценить как и обусловлена она тем, что для поиска и центра, и радиуса можно использовать алгоритм Дейкстры[8].

Также могут существовать и другие решения. Для их поиска создадим полиноминальный алгоритм, который будет перебирать все возможные решения для заданного количества стражей.

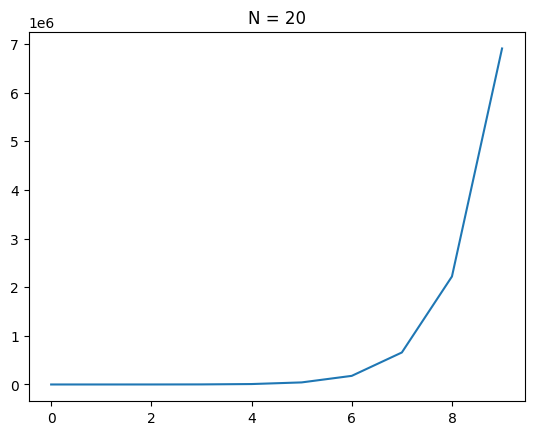


Рисунок 2. График роста количества комбинаций от радиуса

Будем формировать строку длины состоящую из вершин графа . Далее считаем количество вхождений каждой вершины в строку, тогда количество строк равно числу сочетаний с повторениями . Так мы получим все возможные комбинации вершин и расположения их стражей и нам останется проверить, является ли данная комбинация решением. Сложность такого алгоритма напрямую зависит от связности графа и экспоненциально растет при увеличении радиуса (Рисунок 2).

Напишем алгоритм, который будет решать задачу вышеописанным способом (Алгоритм №2).

1. Вычисляем радиус графа.
2. Генерируем все возможные комбинации расположения стражей.
3. Для каждой комбинации вершин выполняем следующие шаги:
   * Создаем словарь, где ключами являются вершины, а значениями - количество их вхождений в комбинацию.
   * Инициализируем список цветов вершин, где все вершины изначально помечены как синие.
   * Для каждой вершины с количеством вхождений больше 1 создаем ограниченное по глубине дерево обхода в ширину и окрашиваем эти вершины в красный цвет, а их соседей - в зеленый цвет.
   * Если в итоговом списке цветов отсутствуют синие вершины, то считаем комбинацию доминирующим множеством и добавляем ее в список решений.

Алгоримт написан на языке Python с использованием библиотеки для работы с графами Networkx и модуль Itertools, в котором содержатся реализации комбинаторных функций.

## **Поиск альтернативных решений**

При экспериментах ни разу не было встречено решение, в котором количество вершин со стражами превышало бы 2. Можно предположить, что при число вершин со стражами не может превышать 2. Попробуем доказать это утверждение.

Пусть **v1** – центральная вершина графа **G1**, **v2** – центральная вершина графа **G2**, возьмем вершину **с1**, располагающуюся на кратчайшем пути между **v1** и **v2** и находящуюся на расстоянии **R2** от **v1**, тогда расстояние между **с1** и **v2** не превышает . Возьмем произвольную вершину **x** из графа **G1**, поскольку расстояние от **x** до **v1** не превышает **R1**, то в графе **G** существует путь из **с1** в **x (**через **v1)** длиныне более . Аналогично для любой вершины **y** из графа **G2** расстояние от **с1** до **y** в графе **G** не превышает .

Таким образом эксцентриситет вершины **с1** в графе **G** не превышает , следовательно и радиус не превышает . Нетрудно понять, что расстояние от **с1** до наиболее удаленной вершины в **G1** не превышает и в случае с графом **G2**, аналогичное рассуждение можно провести для **с2**.

Тогда мы можем составить неравенство , при этом мы можем составить равенство , которое следует из того, что для существования решения число доминации полученного графа должно быть сумме чисел доминаций включенных в него **G1** и **G2**. Из этого следует, что необходимым условием существования покрытия графа из 2 вершин является наличие в нем 2 подграфов, включающих в себя все вершины графа и удовлетворяющих выражению .

Теперь докажем, что не может существовать покрытия более чем из 2 вершин.

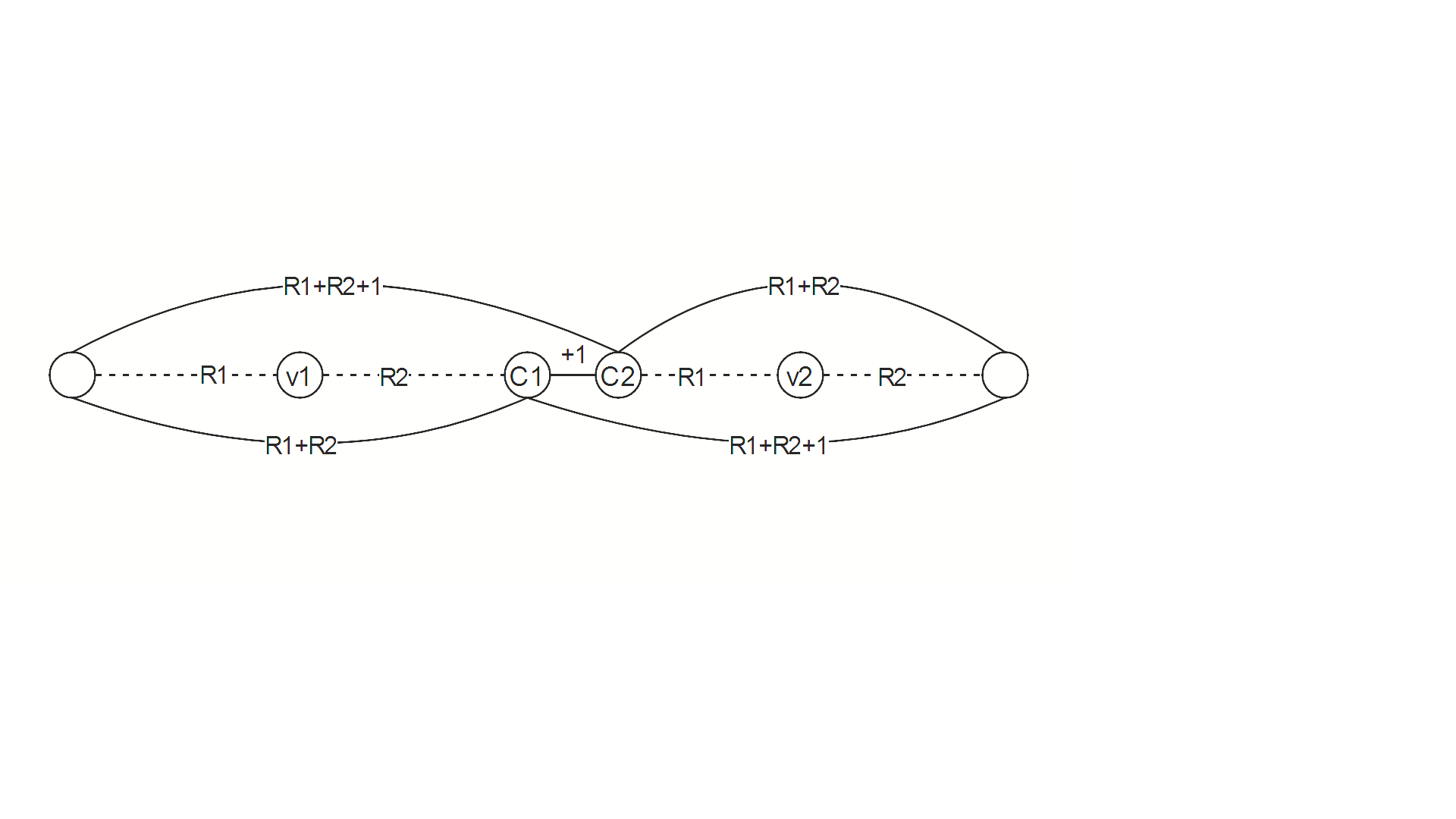


Рисунок 3. Графы G1 и G2

Представим, что граф с покрытием из 2 вершин состоит из подграфов **G1** и **G2**, соединенных так, чтобы кратчайший путь между центрами графов было равно (Рисунок 3). Снова возьмем вершину **с1** расположенною на кратчайшем пути между центрами **v1**  и **v2** на расстоянии **R2** от **v1** и смежную с ней вершину **с2**,расположенную на расстоянии **R1** от **v2**.

Предположим, что для **с1** расстояние до максимально удаленной вершины в **G1** равно , а в **G2** . Тогда **с1** и **с2** являются центром графа и выполняется неравенство , в ином случае равенство не выполняется.

Возьмем уже наш граф с центрами **с1** и **с2**, радиусом **R1,2** и добавим граф **G3** и радиусом **R3**, так, чтобы путь из **с1** к **v3** проходил через **с2** и был равен (Рисунок 4).

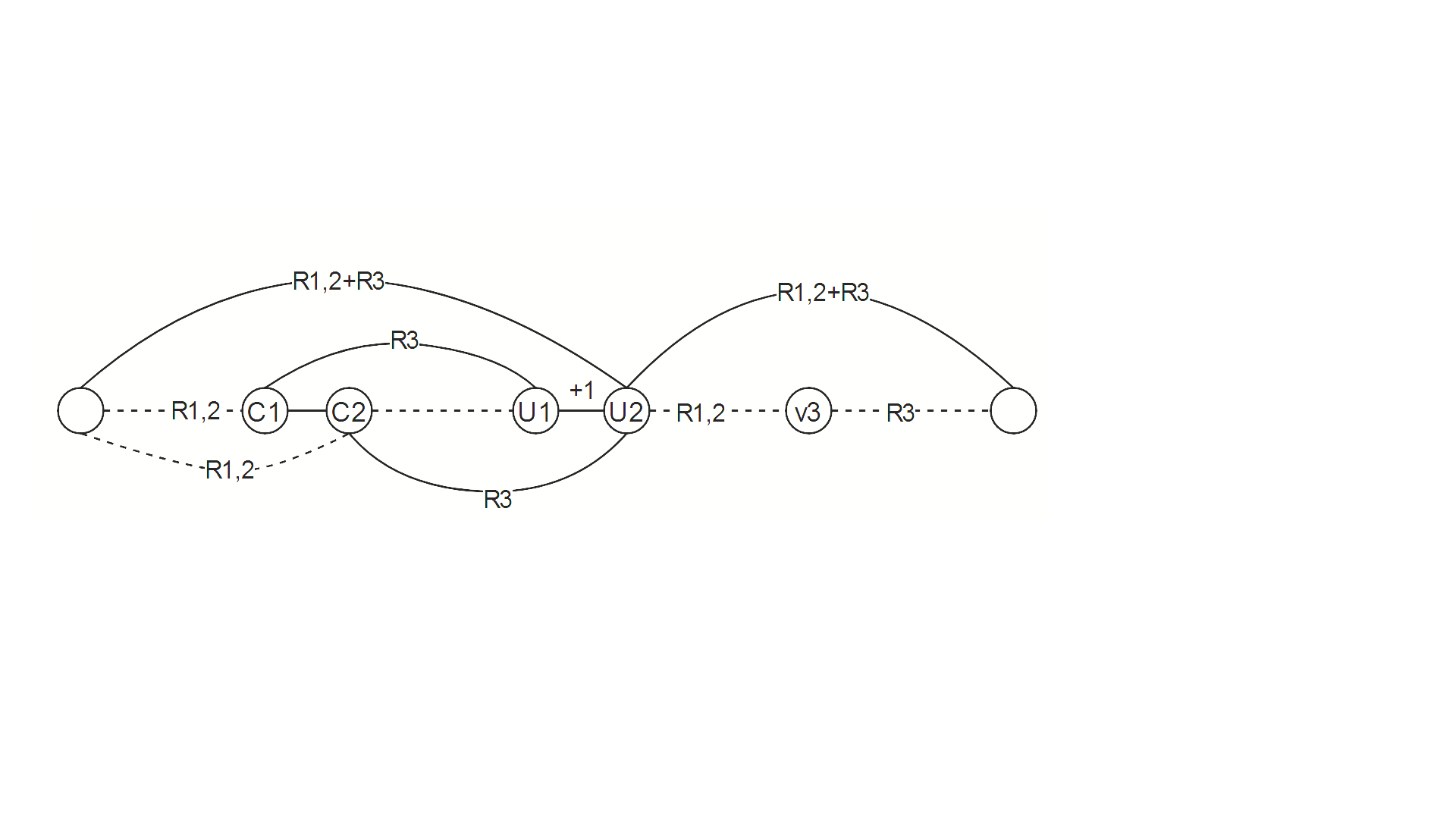


Рисунок 4. Графы G1,2 и G3

Возьмем вершину **u1**, располагающуюся на пути между **с1** и **v3**, и находящуюся на расстоянии **R3** от **с1**. Т.к. **с2** лежит между **с1** и **v3** то расстояние до **u2** смежной с **u1** также будет равно **R3**. Из всего вышеперечисленного следует, что расстояние до крайних точек обоих графов для вершины **u2** не превышает , что не соответствует необходимому условию существования решения, которое мы вывели выше.

Из приведенных выше рассуждений можно понять, что для задачи **mRDF** существует 2 типа решений - расположение стража в центре или же поиск комбинации расположения стражей в 2 вершинах. Необходимым условием существования решения с 2 вершинами является наличие в графе более чем одного центра.

## **Оптимизация алгоритма решения**

На основе выводов, сделанных в прошлой части, можно создать новый алгоритм, который будет более эффективно искать решения нашей задачи.

1. Определяется радиус графа - максимальное расстояние между вершинами.
2. Для каждой пары вершин графа:
   * Если вершины совпадают, то:
     + Создается ограниченное по глубине дерево обхода в ширину от одной из вершин с глубиной радиуса.
     + Окрашивается данная вершина в красный, а достижимые из нее вершины - в зеленый.
     + Если все вершины графа окрашены в красный или зеленый цвет, это является решением задачи.
   * Если вершины отличаются, то:
     + Проводятся итерации от 1 до радиуса графа:
     + Создается ограниченное по глубине дерево обхода в ширину от одной из вершин с текущей глубиной.
     + Создается ограниченное по глубине дерево обхода в ширину от другой вершины с глубиной, равной разности радиуса и текущей глубины.
     + Окрашиваются вершины обоих деревьев в соответствующие цвета.
   * Если все вершины графа окрашены в красный или зеленый цвет, это является решением задачи.
3. Выводятся результаты и время выполнения алгоритма.

Алгоритм итеративно проверяет все возможные комбинации стражей, учитывая радиус графа. Основная проверка осуществляется через раскраску вершин в графе: красный цвет обозначает присутствие стражей, зеленый – вершина покрыта, а синий – непокрытые вершины. Алгоритм завершается, проверены все возможные комбинации стражей.

## **Эксперименты с полученными алгоритмами**

Проведем сравнение двух алгоритмов. Следующая серия тестов проведена для лестничных графов.

Лестничный граф - это особый тип графа, который имеет структуру, напоминающую лестницу. Он состоит из двух рядов вершин, связанных между собой горизонтальными и вертикальными ребрами.

Обычно лестничный граф представляется в виде последовательности вершин, где каждая вершина из одного ряда соединена ребром с соответствующей вершиной в другом ряду, а также с соседними вершинами в том же ряду. В результате получается структура, которая напоминает лестницу или ступеньки.

На оси X отображено число ступеней, т.е. число вершин в 2 раза больше, в лестничных графах отношение числа вершин к радиусу примерно 4:1, что сильно замедляет работу алгоритма, также как и большое количество циклов.

|  |  |
| --- | --- |
|  |  |
| Рисунок 5. Время работы Алгоритма №2 | Рисунок 6. Время работы Алгоритма №3 |
|  |  |
| Рисунок 7. Кол-во итераций Алгоритма №2 | Рисунок 8. Кол-во итераций Алгоритма №3 |
|  |  |

По графикам видно, что алгоритм №3 эффективнее алгоритма №2, также по рисункам 5, 7 видно, что время работы и сложность алгоритма алгоритма №2 экспоненциально растет. Тесты не проводились для значений более 14, т.к. это приводило к переполнению оперативной памяти. В это же время кажется, что сложность алгоритма №3 растет линейно, но это не так, что мы можем видеть на Рисунках 9, 10.

|  |  |
| --- | --- |
|  |  |
| Рисунок 9. Время работы Алгоритма №3 | Рисунок 10. Кол-во итераций Алгоритма№3 |

Сложность алгоритма №3 можно оценить как O(n5) в худшем случае и складывается она из  **–** количество итераций по вершинам графа и – сложность алгоритма поиска в ширину, применяемого для поиска покрываемых вершин. Поэтому мы можем видеть, что время работы алгоритма.

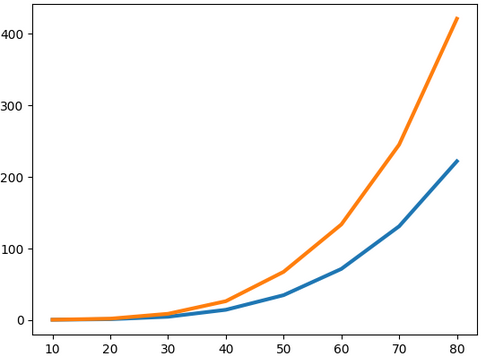


Рисунок 11. Сравнение времени работы для пути и лестницы

На рисунке 11 мы можем видеть время работы алгоритма №3 для лестничного графа и пути. При равном количестве вершин время работы для пути растет быстрее, т.к. в путях самое высокое отношение радиуса к числу вершин, примерно 1:2.

Теперь проведем серию тестов на графах GDM(Dorogovtsev-Goltsev-Mendes). Такие графы строятся алгоритмом, который начинается с создания трех узлов и трех ребер, образуя треугольник, а затем добавляет один узел за раз. Каждый раз, когда добавляется узел, случайным образом выбирается ребро, и к узлу добавляются два новых ребра, соединяющих его с двумя конечностями выбранного ребра.

Этот процесс генерирует граф с распределением степеней, подчиняющимся степенному закону, так как узлы с большим количеством ребер имеют больше шансов быть выбранными, поскольку их ребра чаще представлены в множестве ребер.

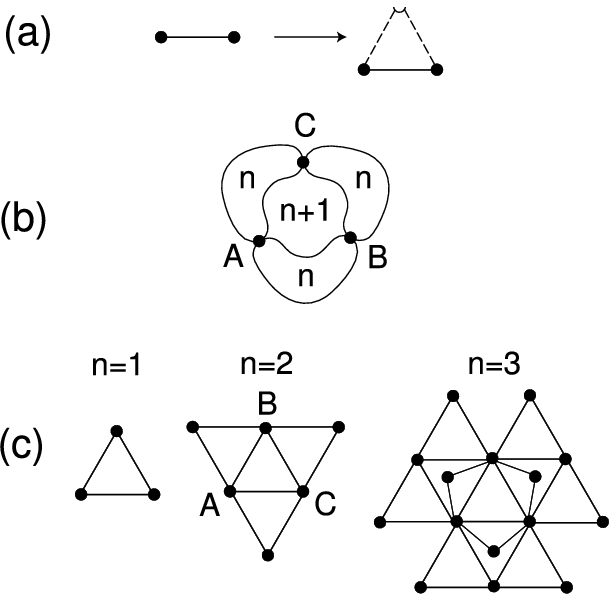


Рисунок 12. Графы GDM

Алгоритм всегда создает планарные графы. Такой граф хорошо подойдет нам для понимания, как ведет себя алгоритм №3 на обычных графах.

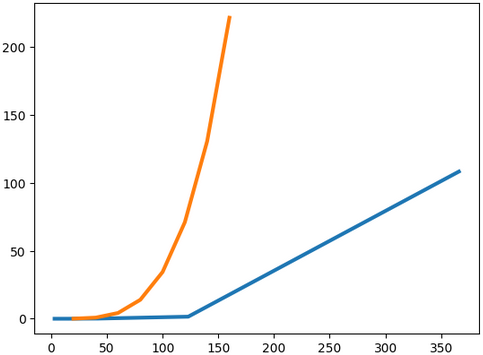


Рисунок 13. Сравнение времени работы для лестницы и GDM

По рисунку 13 можно наблюдать, что время работы напрямую зависит от типа графа, а в частности от его радиуса.

Преимуществом алгоритма №2 является то, что его можно легко модифицировать для поиска числа доминации с любыми ограничениями на дистанцию, но резко возрастающая сложность не дает применять его на больших графах.

# Заключение

В данной дипломной работе были рассмотрены классическая и модифицированная задачи римского доминирования. Задача римского доминирования является важной исследовательской темой в области теории графов и оптимизации. Она находит применение в различных практических сферах, включая планирование сетей связи, размещение сенсорных узлов, определение мощности сигнала и другие.

В ходе работы была проведена обзорная аналитическая часть, в которой были рассмотрены основные понятия и определения, связанные с задачей римского доминирования. Были изучены существующие подходы и алгоритмы для ее решения.

Далее была представлена классическая задача римского доминирования, в которой требуется найти минимальное доминирующее множество вершин графа. Был рассмотрен жадный алгоритм, основанный на выборе вершин с наибольшим числом неосвещенных соседей. Этот алгоритм позволяет найти приближенное решение задачи, однако не гарантирует оптимальность.

Затем была представлена модифицированная задача римского доминирования, в которой отсутствует ограничение на размещение стражей в одной вершине. Был предложен новый алгоритм, основанный на переборе всех возможных комбинаций вершин и проверке их доминирования. Этот алгоритм позволяет найти все доминирующие множества графа, включая непересекающиеся.

В заключение, классическая задачиf римского доминирования и ее модификации представляют собой важные задачи в области теории графов. Жадный алгоритм для классической задачи обеспечивает приближенное решение, а новый алгоритм для модифицированной задачи позволяет найти все доминирующие множества графа. Дальнейшие исследования могут быть направлены на разработку алгоритма поиска решения для ограниченной дистанции и изучение приложений задачи римского доминирования в различных областях.

# Список литературы

1. Stewart I. Defend the Roman empire! //Scientific American. – 1999. – Т. 281. – №. 6. – С. 136-138.
2. E. J. Cockayne, P. A. Dreyer, S. M. Hedetniemi, and S. T. Hedetniemi, “Roman domination in graphs,” Discrete Mathematics, vol. 278, no. 1, pp. 11–22, 2004.
3. A Game-Theoretic Approach to Solving the Roman Domination Problem Xiuyang Chen, Changbing Tang Member, IEEE, Zhao Zhang Member, IEEE, and Guanrong Chen Life Fellow, IEEE Feb 2023
4. Shang W., Hu X. The roman domination problem in unit disk graphs //Computational Science–ICCS 2007: 7th International Conference, Beijing, China, May 27-30, 2007, Proceedings, Part III 7. – Springer Berlin Heidelberg, 2007. – С. 305-312.
5. Wang L. et al. Approximation algorithm for a generalized Roman domination problem in unit ball graphs //Journal of Combinatorial Optimization. – 2020. – Т. 39. – №. 1. – С. 138-148.
6. Pagourtzis A. et al. Server placements, Roman domination and other dominating set variants //Foundations of Information Technology in the Era of Network and Mobile Computing: IFIP 17 th World Computer Congress—TC1 Stream/2 nd IFIP International Conference on Theoretical Computer Science (TCS 2002) August 25–30, 2002, Montréal, Québec, Canada. – Springer US, 2002. – С. 280-291.
7. Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн Алгоритмы: построение и анализ — 3-е изд. — М.: «Вильямс», 2013. — с. 696-701.
8. Dijkstra L., Gagarin A., Zverovich V. Weighted domination models and randomized heuristics //arXiv preprint arXiv:2203.00799. – 2022.

# Исходный код

Алгоритм №1 Жадный алгоритм для классической задачи

import networkx as nx

import matplotlib.pyplot as plt

def uncover\_neighbours(G, colors, g):

  r = 0

  node = G[g]

  for i in node:

    if colors[i] == 'black':

      r = r + 1

  return r

yR = 0

N = 10

graph = {

    0: [7, 8, 9],

    1: [2, 3, 4],

    2: [1, 3, 4, 5],

    3: [1, 2, 5, 6],

    4: [1, 2, 5, 6],

    5: [2, 3, 4, 7, 8],

    6: [3, 4, 7,  9],

    7: [ 5, 6, 9, 0],

    8: [5, 6, 9, 0],

    9: [6, 7, 0]

}

colors = ['black']\*N

G = nx.Graph(graph)

while('black' in colors):

  max\_node = -1

  max\_count = -1

  for i in G:

    if colors[i] == 'black':

      if uncover\_neighbours(G, colors, i) > max\_count:

        max\_node = i

        max\_count = uncover\_neighbours(G, colors, i)

  if max\_count > 0:

    colors[max\_node] = 'red'

    nodes = G[max\_node]

    for i in nodes:

      if colors[i] == 'black':

        colors[i] = 'blue'

  else:

    colors[max\_node] = 'green'

pos = nx.spring\_layout(G)

nx.draw(G, pos, node\_color=colors, with\_labels=True)

plt.show()

Алгоритм №2 Полный перебор всех комбинаций для модифицированной задачи

from itertools import \*

import networkx as nx

import matplotlib.pyplot as plt

import numpy as np

from datetime import datetime

G = nx.ladder\_graph(12)

radius = nx.radius(G)

print(radius)

center = nx.center(G)

N = len(G)

print(N)

letters = list(range(0, N))

start\_time = datetime.now()

r = 0

res = []

comb = list(combinations\_with\_replacement(letters, radius+1))

print(len(comb))

for i in comb:

  tmp = {}

  for k in i:

    if tmp.get(k) == None:

      tmp[k] = 0

    tmp[k] = tmp[k]+1

  key = list(tmp.keys())

  colors = ['blue']\*N

  for x in tmp:

    if tmp[x]>1:

      G\_tmp = nx.bfs\_tree(G, source=x, depth\_limit=tmp[x]-1)

    else:

      G\_tmp = x

      colors[x] = 'red'

      break

    for node in G\_tmp.nodes():

      r = r+1

      if tmp.get(node) != None:

          colors[node] = 'red'

          G.nodes[node]['label'] = str(tmp[node])

      else:

          if colors[node] == 'blue':

            colors[node] = 'green'

  if 'blue' not in colors:

    pos = nx.spring\_layout(G)

    nx.draw(G, pos, node\_color=colors, with\_labels=True)

    plt.title(tmp)

    print(tmp)

    plt.show()

    res.append(tmp)

print(datetime.now()-start\_time)

Алгоритм №3 Перебор комбинаций пар вершин для модифицированной задачи

from itertools import \*

import networkx as nx

import matplotlib.pyplot as plt

import numpy as np

from datetime import datetime

num = 10

G = nx.ladder\_graph(100)

#G = nx.path\_graph(16)

radius = nx.radius(G)

print(radius)

center = nx.center(G)

N = len(G)

print(N)

letters = list(range(0, N))

start\_time = datetime.now()

r = 0

res = []

for i in range(len(G)):

  for j in range(i, len(G)):

    if i == j:

      colors = ['blue']\*N

      G\_tmp = nx.bfs\_tree(G, source=i, depth\_limit = radius)

      colors[i] = 'red'

      for g in G\_tmp:

        if colors[g] == 'blue':

          colors[g] = 'green'

      if 'blue' not in colors:

        res.append(dict([(i, radius + 1)]))

        pos = nx.spring\_layout(G)

        nx.draw(G, pos, node\_color=colors, with\_labels=True)

        plt.title(dict([(i , radius + 1)]))

        plt.show()

    else:

      for k in range(1, radius + 1):

        colors = ['blue']\*N

        if k>1:

          G\_tmp1 = nx.bfs\_tree(G, source=i, depth\_limit=k-1)

          colors[i] = 'red'

          for g in G\_tmp1:

            if colors[g] == 'blue':

              colors[g] = 'green'

        else:

          G\_tmp1 = i

          colors[i] = 'red'

        if radius - k + 1 > 1:

          G\_tmp2 = nx.bfs\_tree(G, source=j, depth\_limit = radius - k)

          colors[j] = 'red'

          for g in G\_tmp2:

            if colors[g] == 'blue':

              colors[g] = 'green'

        else:

          G\_tmp2 = i

          colors[j] = 'red'

        if 'blue' not in colors:

          res.append(dict([(i, k), (j, radius-k+1)]))

          pos = nx.spring\_layout(G)

          nx.draw(G, pos, node\_color=colors, with\_labels=True)

          plt.title(dict([(i, k), (j, radius-k+1)]))

          plt.show()

print(datetime.now()-start\_time)

print(res)