Федеральное государственное автономное образовательное учреждение высшего образования
Санкт-Петербургский Политехнический университет Петра Великого
Физико-Механический институт

Лабораторная 4 – Циклы и раскраска

«Вариант 4 – Хроматическое число графа»

Выполнил студент гр. 5030102/20101: Бугайцев М.В. _____
Преподаватель: Новиков Ф. А. _____
Работа принята: Дата _____

Санкт-Петербург
2024

Содержание

1 Введение

1.1 Цели и задачи лабораторной работы:

На вход программе подаётся граф. Требуется найти, используя приближённый алгоритм, и вывести хроматическое число данного графа. Хроматическое число — это минимальное количество цветов, необходимое для раскраски вершин графа так, чтобы никакие две смежные вершины не имели одинакового цвета.

Цель данной лабораторной работы заключается в изучении методов приближённого решения задачи раскраски графов и оценке их эффективности. В частности, мы будем использовать жадный алгоритм для нахождения хроматического числа и анализировать его точность.

Важной задачей является выяснение, всегда ли выбранный алгоритм будет давать точный ответ. Если да, то необходимо объяснить, почему это так. Если нет, то следует привести контрпример, демонстрирующий случаи, когда алгоритм может ошибаться.

В рамках работы будут рассмотрены следующие задачи:

1. Изучение основных понятий, связанных с графами и их раскраской.
2. Описание алгоритма, используемого для нахождения хроматического числа.
3. Реализация графа и его методов в программном обеспечении.
4. Проведение экспериментов для проверки работы алгоритма на различных типах графов.
5. Анализ результатов и выводы о точности приближённого алгоритма.

2 Теоретическая часть

2.1 Основные понятия графов:

Графом называется пара $G = (V, E)$, где V — множество вершин, а E — множество рёбер, соединяющих пары вершин. Важным понятием является хроматическое число графа, которое обозначается как $\chi(G)$ и представляет собой минимальное количество цветов, необходимых для раскраски вершин графа так, чтобы никакие две смежные вершины не имели одинакового цвета.

2.2 Описание алгоритма:

Для нахождения хроматического числа графа можно использовать приближённый алгоритм, основанный на жадном методе. Алгоритм работает следующим образом:

1. Инициализируем пустое множество цветов.
2. Для каждой вершины графа:
   - Находим все цвета, которые уже используются соседними вершинами.
   - Присваиваем текущей вершине наименьший доступный цвет, который не используется её соседями.
3. После обработки всех вершин, количество использованных цветов будет являться приближённым значением хроматического числа графа.

Этот алгоритм не гарантирует нахождение точного хроматического числа, так как он может не учитывать оптимальные раскраски.

2.3 Сферы применения:

Хроматическое число графа и алгоритмы его нахождения имеют широкий спектр применения в различных областях, включая:

- **Теория графов:** Используется для изучения свойств графов и их структур.
- **Компьютерные науки:** Применяется в задачах, связанных с распределением ресурсов, например, в задачах о раскраске графов, оптимизации сетей и планировании.
- **Операционные исследования:** Используется для решения задач, связанных с оптимизацией, таких как задачи о назначениях и маршрутизации.
- **Социальные сети:** Применяется для анализа взаимодействий между участниками, где вершины представляют пользователей, а рёбра — связи между ними.
- **Биология:** Используется для моделирования взаимодействий между различными биологическими системами, такими как экосистемы или сети метаболизма.
- **Телекоммуникации:** Применяется для оптимизации частотного спектра и управления сетями связи.

3 Практическая часть

3.1 Реализация графа

В данной секции рассматривается реализация не ориентированного графа, основанная на концепции списка смежности. Список смежности представляет собой эффективный способ хранения графа, который позволяет организовать данные о вершинах и их связях в удобной и компактной форме.

3.1.1 Список смежности

Список смежности - это структура данных, в которой каждый узел графа содержит информацию о своих соседях, то есть о других узлах, соединенных с ним ребрами.

В отличие от матрицы смежности, список смежности более экономичен, так как хранит только существующие связи.

Существует также концепция списка рёбер, которая описывает граф в виде набора рёбер, где каждое ребро представлено парой вершин (или тройкой, если граф взвешенный). Список смежности ,более удобен для алгоритмов обхода графа, таких как поиск в глубину (DFS) и поиск в ширину (BFS), так как позволяет быстро находить всех соседей конкретного узла.

3.2 Структура JSON файла

Структура JSON файла, используемого для задания графа, включает два основных элемента: nodes и edges.

- nodes: Список узлов графа. Каждый узел представлен уникальным идентификатором. Например, в следующем примере узлы представлены числами от 1 до 5:

  "nodes": [1, 2, 3, 4, 5]

- edges: Список рёбер графа. Каждое ребро представлено в виде массива, содержащего два узла, которые оно соединяет. Например, в следующем примере рёбра соединяют узлы следующим образом:

  "edges": [[1, 2], [1, 3], [2, 4], [1, 5]]

Таким образом, полный пример JSON файла, описывающего граф, выглядит следующим образом:

```
{
   "nodes": [1, 2, 3, 4, 5],
   "edges": [[1, 2], [1, 3], [2, 4], [1, 5]]
}
```

4 Сложности операций в классе Graph

4.1 Инициализация графа (__init__)

Метод __init__ отвечает за создание нового экземпляра класса Graph. При его вызове инициализируется пустой граф, представленный в виде словаря, где ключами являются узлы, а значениями — списки смежных узлов.

- **Временная сложность:** $O(n + m)$, где n — количество узлов, а m — количество рёбер.
- **Описание:** Метод принимает два необязательных параметра: nodes и edges. Если nodes не равен None, то для каждого узла в списке вызывается метод add_node, который добавляет узел в граф. Если edges также не равен None, то для каждой пары узлов (u, v) в списке рёбер вызывается метод add_edge, который добавляет рёбра между узлами u и v. Таким образом, граф может быть инициализирован сразу с набором узлов и рёбер, что упрощает его создание и позволяет избежать дополнительных вызовов методов после инициализации.

4.2 Добавление ребра (add_edge)

Метод add_edge отвечает за добавление ребра между двумя узлами графа. При вызове этого метода узлы u и v становятся соседями, и ребро добавляется в обе стороны, так как граф является неориентированным.

- **Временная сложность:** $O(1)$
- **Описание:** Добавление ребра занимает постоянное время, так как операция добавления элемента в список (в данном случае, список смежных узлов) выполняется за $O(1)$. В худшем случае, если узел не существует в графе, он будет создан с пустым списком, что также не влияет на временную сложность.

4.3 Добавление узла (add_node)

Метод add_node отвечает за добавление нового узла в граф. При вызове этого метода, если узел node еще не существует в графе, он добавляется в словарь self.graph с пустым списком смежных узлов.

- **Временная сложность:** $O(1)$
- **Описание:** Добавление узла занимает постоянное время, так как проверка наличия узла в словаре и добавление нового ключа выполняются за $O(1)$. Если узел уже существует, метод просто завершает выполнение без изменений.

4.4 Удаление ребра (remove_edge)

Метод remove_edge отвечает за удаление ребра между двумя узлами графа. При вызове этого метода, если узлы u и v существуют в графе и между ними есть ребро, оно удаляется в обе стороны, так как граф является неориентированным.

- **Временная сложность:** $O(1)$
- **Описание:** Данный метод использует метод discard, который безопасно удаляет элемент из множества за $O(1)$, не вызывая ошибки, если элемент отсутствует. Это позволяет избежать дополнительных проверок на наличие ребра перед его удалением.

4.5 Загрузка графа из JSON (load_tree_from_json)

Метод load_tree_from_json отвечает за загрузку графа из файла в формате JSON. При его вызове происходит чтение данных из указанного файла, после чего граф инициализируется узлами и рёбрами, описанными в загруженных данных.

- **Временная сложность:** $O(n + m)$, где n — количество узлов, а m — количество рёбер.
- **Описание:** Метод принимает один параметр filename, который представляет собой путь к файлу JSON. Внутри метода открывается файл для чтения, и данные загружаются с помощью функции json.load. Ожидается, что загруженные данные содержат два ключа: nodes и edges. Если ключ nodes присутствует, то для каждого узла в списке вызывается метод add_node, который добавляет узел в граф. Аналогично, если ключ edges присутствует, то для каждой пары узлов (u, v) в списке рёбер вызывается метод add_edge, который добавляет рёбра между узлами u и v. Таким образом, метод позволяет легко и быстро загружать граф из внешнего источника, что упрощает работу с графовыми структурами данных.

4.6 Жадная раскраска графа (greedy_coloring)

Метод greedy_coloring отвечает за раскраску графа с использованием жадного алгоритма. При вызове этого метода для каждого узла графа назначается цвет, так что никакие два соседних узла не имеют одинакового цвета.

- **Временная сложность:** $O(n \cdot d)$, где n — количество узлов в графе, а d — максимальная степень узла. В худшем случае для каждого узла необходимо проверить цвета его соседей, что требует линейного времени относительно количества соседей.
- **Сложность по памяти:** $O(n)$, так как необходимо хранить отображение color_map, в котором каждому узлу соответствует его цвет. В дополнение, для хранения цветов соседей используется множество, что также требует линейного пространства в зависимости от количества соседей.
- **Описание:** Метод итерируется по всем узлам графа. Для каждого узла он собирает цвета, уже назначенные его соседям, и находит первый доступный цвет, который не используется соседями. Этот цвет затем назначается текущему узлу. Если узел не имеет соседей или все соседи уже раскрашены, метод завершает выполнение, назначая цвет узлу. В результате получается отображение, где каждому узлу соответствует его цвет.

4.7 Основной метод раскраски графа (graph_coloring)

Метод graph_coloring отвечает за поиск раскраски графа с использованием жадного подхода, пробуя различные количества цветов, начиная с одного и увеличивая их до тех пор, пока не будет найдена допустимая раскраска.

- **Временная сложность:** $O(n^n)$, где n — количество узлов в графе. В худшем случае метод вызывает _graph_coloring для каждого количества цветов от 1 до n, что приводит к экспоненциальной сложности.
- **Сложность по памяти:** $O(n)$, так как необходимо хранить отображение color_map для хранения цветов узлов, а также стек и другие переменные, что требует линейного пространства в зависимости от количества узлов.
- **Описание:** Метод итерируется по количеству цветов от 1 до количества узлов в графе. Для каждого количества цветов _graph_coloring итерируется, чтобы попытаться раскрасить граф. Если раскраска успешна (т.е. метод возвращает color_map), то этот результат возвращается.

4.7.1 Проверка возможность раскраски (_is_safe)

Метод _is_safe отвечает за проверку, возможность ли назначить определённый цвет узлу графа. Этот метод используется в контексте алгоритмов раскраски, чтобы убедиться, что назначаемый цвет не совпадает с цветами соседних узлов.

- **Временная сложность:** $O(d)$, где d — максимальная степень узла. В худшем случае метод проверяет всех соседей узла, чтобы убедиться, что ни один из них не имеет того же цвета.

- **Сложность по памяти:** $O(1)$, так как метод использует только фиксированное количество переменных для хранения состояния и не требует дополнительной памяти, зависящей от размера графа.

- **Описание:** Метод принимает узел, цвет и отображение color_map в качестве аргументов. Он итерируется по всем соседям данного узла и проверяет, есть ли среди них узлы, уже раскрашенные в тот же цвет. Если такой сосед найден, метод возвращает False, указывая на то, что назначение цвета не безопасно. Если же все соседи имеют разные цвета, метод возвращает True, подтверждая, что цвет можно безопасно назначить узлу.

4.7.2 Алгоритм раскраски графа (_graph_coloring)

Метод _graph_coloring реализует алгоритм раскраски графа с использованием DFS. Он пытается назначить цвета узлам графа, соблюдая ограничения на количество доступных цветов.

- **Временная сложность:** $O(m^n)$, где n — количество узлов в графе, а m — максимальное количество доступных цветов. В худшем случае алгоритм может проверить все возможные комбинации раскраски, что приводит к экспоненциальной сложности.

- **Сложность по памяти:** $O(n)$, так как необходимо хранить отображение color_map для хранения цветов узлов, а также стек для хранения состояния узлов и цветов, что требует линейного пространства в зависимости от количества узлов.

- **Описание:** Метод принимает максимальное количество цветов max_colors в качестве аргумента. Он инициализирует пустое отображение color_map и список узлов графа. Затем используется стек для хранения индексов узлов и количества обработанных цветов. В цикле метод извлекает узел из стека и проверяет, был ли он уже раскрашен. Если нет, он пытается назначить узлу цвет, начиная с color_processed. Если цвет может быть присвоен (это проверяется с помощью метода _is_safe), цвет назначается узлу, и в стек добавляются следующие узлы для обработки. Если все узлы успешно раскрашены, метод возвращает color_map. Если раскраска невозможна, возвращается None.

5 Примеры использования

5.1 Неоптимальный результат жадной раскраски

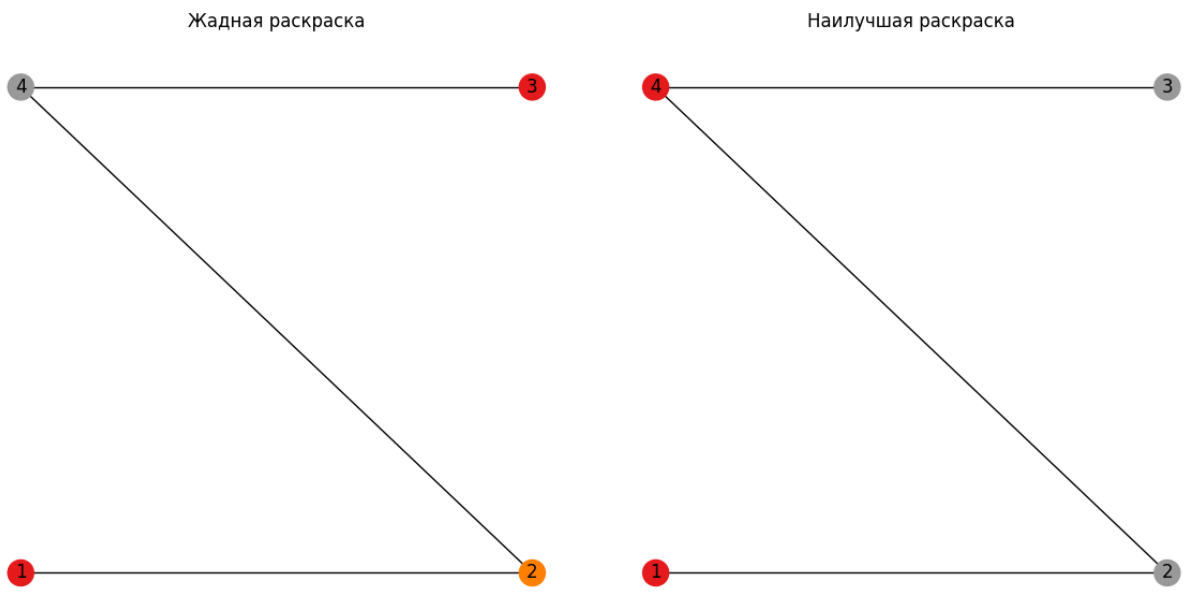

Рис. 1: Пример неоптимального результа жадной раскраск

Как видно на рисунке 1, алгоритм не справляется с задачей в определенных условиях.

Выполнение жадной раскраски

- Обрабатываем узел 1. Соседи: [2]. Нет окрашенных соседей.
- Узел 1 окрашен в цвет 0
- Обрабатываем узел 2. Соседи: [1, 4]. Цвета соседей: 0
- Узел 2 окрашен в цвет 1
- Обрабатываем узел 3. Соседи: [4]. Нет окрашенных соседей.
- Узел 3 окрашен в цвет 0
- Обрабатываем узел 4. Соседи: [2, 3]. Цвета соседей: 0, 1
- Узел 4 окрашен в цвет 2

Раскраска (жадная): "1": 0, "2": 1, "3": 0, "4": 2

Выполнение оптимальной раскраски

Раскраска (оптимальная): "1": 0, "2": 1, "3": 1, "4": 0

Результаты: Хроматическое число (жадная раскраска): 3
Хроматическое число (оптимальная раскраска): 2

5.2 Работа алгоритма на полном графе

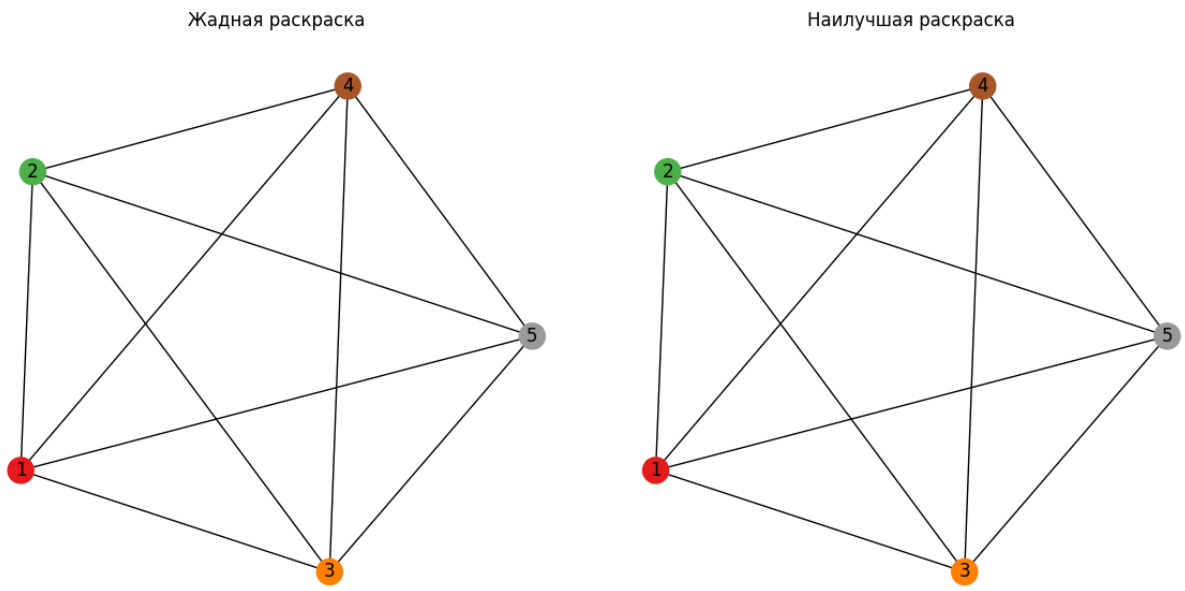

Рис. 2: Результаты работы алгоритма на полном графе

Выполнение жадной раскраски

- Обрабатываем узел 1. Соседи: [2, 3, 4, 5]. Нет окрашенных соседей.
- Узел 1 окрашен в цвет 0
- Обрабатываем узел 2. Соседи: [1, 3, 4, 5]. Цвета соседей: 0
- Узел 2 окрашен в цвет 1
- Обрабатываем узел 3. Соседи: [1, 2, 4, 5]. Цвета соседей: 0, 1
- Узел 3 окрашен в цвет 2
- Обрабатываем узел 4. Соседи: [1, 2, 3, 5]. Цвета соседей: 0, 1, 2
- Узел 4 окрашен в цвет 3
- Обрабатываем узел 5. Соседи: [1, 2, 3, 4]. Цвета соседей: 0, 1, 2, 3
- Узел 5 окрашен в цвет 4

Раскраска (жадная): "1": 0, "2": 1, "3": 2, "4": 3, "5": 4

Выполнение оптимальной раскраски

Раскраска (оптимальная): "1": 0, "2": 1, "3": 2, "4": 3, "5": 4

Результаты: Хроматическое число (жадная раскраска): 5
Хроматическое число (оптимальная раскраска): 5

5.3 Работа алгоритма на изоморфном графе

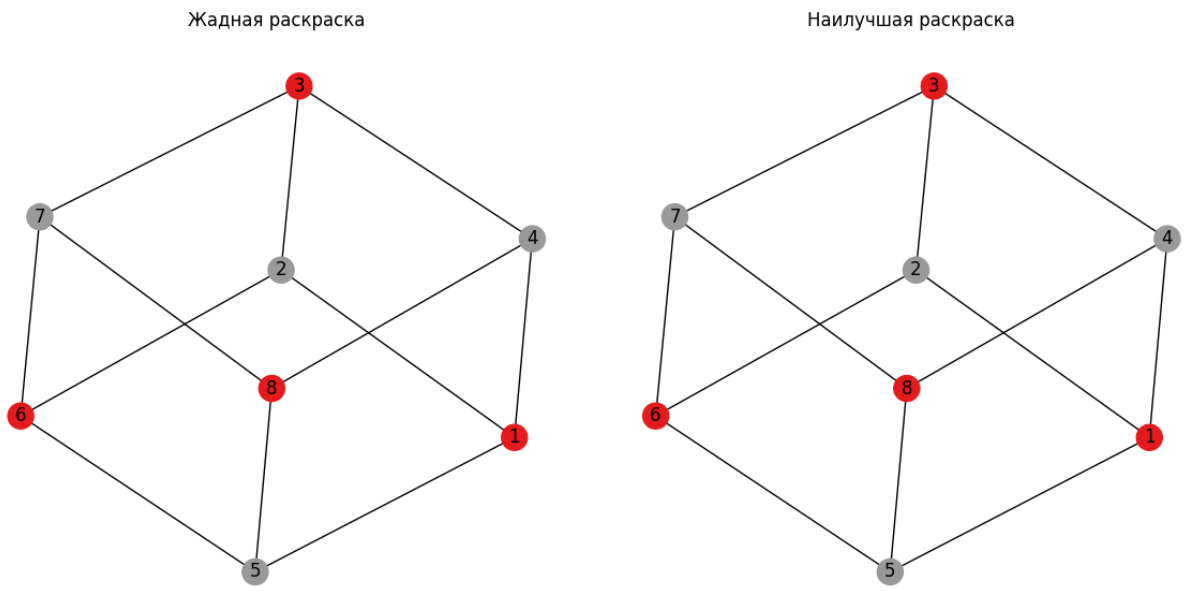

Рис. 3: Результаты работы алгоритма на изоморфном графе

Выполнение жадной раскраски

- Обрабатываем узел 1. Соседи: [2, 4, 5]. Нет окрашенных соседей.
- Узел 1 окрашен в цвет 0
- Обрабатываем узел 2. Соседи: [1, 3, 6]. Цвета соседей: 0
- Узел 2 окрашен в цвет 1
- Обрабатываем узел 3. Соседи: [2, 4, 7]. Цвета соседей: 1
- Узел 3 окрашен в цвет 0
- Обрабатываем узел 4. Соседи: [8, 1, 3]. Цвета соседей: 0
- Узел 4 окрашен в цвет 1
- Обрабатываем узел 5. Соседи: [8, 1, 6]. Цвета соседей: 0
- Узел 5 окрашен в цвет 1
- Обрабатываем узел 6. Соседи: [2, 5, 7]. Цвета соседей: 1
- Узел 6 окрашен в цвет 0
- Обрабатываем узел 7. Соседи: [8, 3, 6]. Цвета соседей: 0
- Узел 7 окрашен в цвет 1
- Обрабатываем узел 8. Соседи: [4, 5, 7]. Цвета соседей: 1
- Узел 8 окрашен в цвет 0

Раскраска (жадная): "1": 0, "2": 1, "3": 0, "4": 1, "5": 1, "6": 0, "7": 1, "8": 0

Выполнение оптимальной раскраски

Раскраска (оптимальная): "1": 0, "2": 1, "3": 0, "4": 1, "5": 1, "6": 0, "7": 1, "8": 0

Результаты: Хроматическое число (жадная раскраска): 2
Хроматическое число (оптимальная раскраска): 2

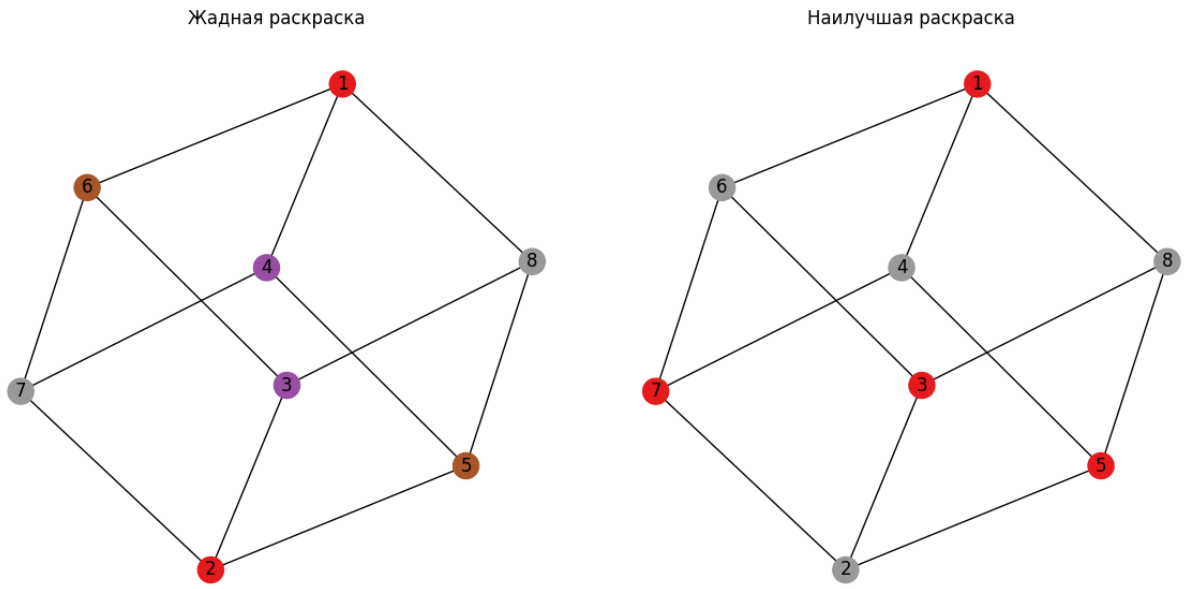

Рис. 4: Результаты работы алгоритма на изоморфном графе

Выполнение жадной раскраски

- Обрабатываем узел 1. Соседи: [8, 4, 6]. Нет окрашенных соседей.
- Узел 1 окрашен в цвет 0
- Обрабатываем узел 2. Соседи: [3, 5, 7]. Нет окрашенных соседей.
- Узел 2 окрашен в цвет 0
- Обрабатываем узел 3. Соседи: [8, 2, 6]. Цвета соседей: 0
- Узел 3 окрашен в цвет 1
- Обрабатываем узел 4. Соседи: [1, 5, 7]. Цвета соседей: 0
- Узел 4 окрашен в цвет 1
- Обрабатываем узел 5. Соседи: [8, 2, 4]. Цвета соседей: 0, 1
- Узел 5 окрашен в цвет 2
- Обрабатываем узел 6. Соседи: [1, 3, 7]. Цвета соседей: 0, 1
- Узел 6 окрашен в цвет 2
- Обрабатываем узел 7. Соседи: [2, 4, 6]. Цвета соседей: 0, 1, 2
- Узел 7 окрашен в цвет 3
- Обрабатываем узел 8. Соседи: [1, 3, 5]. Цвета соседей: 0, 1, 2
- Узел 8 окрашен в цвет 3

Раскраска (жадная): "1": 0, "2": 0, "3": 1, "4": 1, "5": 2, "6": 2, "7": 3, "8": 3

Выполнение оптимальной раскраски

Раскраска (оптимальная): "1": 0, "2": 1, "3": 0, "4": 1, "5": 0, "6": 1, "7": 0, "8": 1

Результаты: Хроматическое число (жадная раскраска): 4
Хроматическое число (оптимальная раскраска): 2

6 Интерфейс приложения

6.1 Пример запуска

Приложение можно запустить из командной строки с использованием следующих аргументов:

- –nodes - список узлов графа в формате: узел1.
- –edges - список рёбер графа в формате: узел1 узел2.
- –json-load - загрузка графа из указанного JSON файла.

6.1.1 Задание рёбер графа

В этом примере мы задаем граф, указывая его рёбра через параметр –edges. Узлы графа будут автоматически добавлены на основе указанных рёбер.

python main.py --edges A B C D E F

6.1.2 Задание узлов и рёбер графа

Если необходимо явно указать узлы графа, можно использовать параметр –nodes вместе с –edges.

python main.py --nodes A B C D --edges A B C D

В этом примере мы создаем граф с узлами A, B, C и D, а также рёбрами A-B и C-D.

6.1.3 Загрузка графа из JSON файла

Если граф уже сохранен в формате JSON, его можно загрузить с помощью параметра –json-load.

python main.py --json-load graph.json

7 Заключение

В ходе выполнения лабораторной работы были изучены основные методы приближённого решения задачи раскраски графов, а также реализован жадный алгоритм для нахождения хроматического числа. Проведённые эксперименты показали, что выбранный алгоритм может давать приближённые результаты, которые, однако, не всегда являются точными.

Важно отметить, что жадный алгоритм может давать разные результаты для изоморфных графов. Это связано с тем, что выбор цвета для каждой вершины могут варьироваться в зависимости от порядка их обхода. Таким образом, несмотря на то, что изоморфные графы имеют одинаковую структуру, жадный алгоритм может присвоить им разные хроматические числа, что подчеркивает его приближённый характер и зависимость от порядка обработки вершин.

8 Приложение

В этом разделе приведен код программы.

```
from collections import defaultdict, deque
import matplotlib.pyplot as plt
import networkx as nx
import argparse
import json
import sys


class RedirectPrint:
    def __init__(self, filename: str):
        self.filename = filename
        self.original_stdout = sys.stdout

    def __enter__(self):
        self.file = open(self.filename, 'w')
        sys.stdout = self.file
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        sys.stdout = self.original_stdout
        self.file.close()
        with open(self.filename, 'r') as f:
            print(f.read())


class Graph:
    def __init__(self, nodes=None, edges=None) -> None:
        self.graph = defaultdict(set)
        if nodes is not None:
            for node in nodes:
                self.add_node(node)
        if edges is not None:
            for u, v in edges:
                self.add_edge(u, v)

    def add_edge(self, u, v) -> None:
        self.graph[u].add(v)
        self.graph[v].add(u)

    def add_node(self, node) -> None:
        if node not in self.graph:
            self.graph[node] = set()

    def remove_edge(self, u, v) -> None:
        self.graph[u].discard(v)
        self.graph[v].discard(u)

    def greedy_coloring(self) -> dict:
        color_map = {}

        for node in self.graph:
            neighbor_colors = {
                color_map[neighbor] for neighbor in self.graph[node]
                if neighbor in color_map}

            color = 0
            while color in neighbor_colors:
                color += 1

            color_map[node] = color

        return color_map

    def _is_safe(self, node, color, color_map):
        for neighbor in self.graph[node]:
```

```
            if neighbor in color_map and color_map[neighbor] == color:
                return False
        return True

    def _graph_coloring(self, max_colors):
        color_map = {}
        nodes = list(self.graph.keys())
        stack = deque([(0, 0)])
        while stack:
            node_index, color_processed = stack.pop()
            if node_index == 0 and color_processed == 1:
                return None

            if node_index == len(nodes):
                return color_map

            node_name = nodes[node_index]
            if nodes[node_index] in color_map:
                del color_map[node_name]

            for color in range(color_processed, max_colors):
                if self._is_safe(
                        node_name, color, color_map):
                    color_map[node_name] = color
                    stack.append((node_index, color + 1))
                    stack.append((node_index + 1, 0))
                    break
        return None

    def graph_coloring(self):
        for num_colors in range(1, len(self.graph) + 1):
            color_map = self._graph_coloring(num_colors)
            if color_map:
                return color_map
        return None

    def draw(self, *, greedy_coloring: dict, best_coloring: dict) -> None:
        G = nx.Graph()
        for n in self.graph.keys():
            G.add_node(n)
        for u in self.graph:
            for v in self.graph[u]:
                G.add_edge(u, v)
        fig, axs = plt.subplots(1, 2, figsize=(12, 6))

        colors_greedy = [greedy_coloring[node] for node in G.nodes()]
        pos = nx.spring_layout(G)
        nx.draw(
            G,
            pos,
            with_labels=True,
            node_color=colors_greedy,
            cmap=plt.get_cmap('Set1'),
            ax=axs[0]
        )
        axs[0].set_title("Жадная раскраска")

        colors_best = [best_coloring[node] for node in G.nodes()]
        nx.draw(
            G,
            pos,
            with_labels=True,
            node_color=colors_best,
            cmap=plt.get_cmap('Set1'),
            ax=axs[1]
        )
        axs[1].set_title("Наилучшая раскраска")

        plt.tight_layout()
```

```
135         plt.show()
136
137     def load_tree_from_json(self, filename: str):
138         with open(filename, 'r') as f:
139             graph_data = json.load(f)
140
141         nodes = graph_data.get('nodes', [])
142         edges = graph_data.get('edges', [])
143         for node in nodes:
144             self.add_node(node)
145         for u, v in edges:
146             self.add_edge(u, v)
147
148
149 def main():
150     parser = argparse.ArgumentParser(description="Хроматическое число графа")
151     graph_group = parser.add_argument_group('Граф', 'Способы задания графа')
152     graph_group.add_argument(
153         '--edges',
154         nargs='+',
155         help='Список рёбер графа в формате: узел1 узел2 например(: A B C D - для рёбер A-B
    и C-D)',
156         required=False)
157     graph_group.add_argument('--nodes', nargs='+',
158                              help='Список узлов графа например(: A B C D)',
159                              required=False)
160     graph_group.add_argument(
161         '--json-load',
162         help='Загрузить граф из JSON файла например(: graph.json)',
163         default='graph.json',
164         required=False)
165
166     args = parser.parse_args()
167     g = Graph()
168     with RedirectPrint('output.txt'):
169         if args.edges:
170             print("Загрузка графа из рёбер.")
171             for node in args.nodes:
172                 g.add_node(node)
173             for i in range(0, len(args.edges), 2):
174                 g.add_edge(args.edges[i], args.edges[i + 1])
175             print(
176                 f"Граф загружен с {len(args.nodes)} узлами и {len(args.edges) // 2}
    рёбрами.")
177         elif args.json_load:
178             print(f"Загрузка графа из файла {args.json_load}.")
179             g.load_tree_from_json(args.json_load)
180             print("Граф успешно загружен из JSON.")
181         else:
182             raise ValueError(
183                 "Неизвестный алгоритм. Укажите рёбра или файл JSON.")
184
185         print("Выполнение жадной раскраски.")
186         greedy_coloring = g.greedy_coloring()
187
188         print(
189             "Раскраска жадная():",
190             json.dumps(
191                 greedy_coloring,
192                 ensure_ascii=False,
193                 indent=4))
194
195         print("Выполнение оптимальной раскраски.")
196         best_coloring = g.graph_coloring()
197
198         print(
199             "Раскраска оптимальная():",
200             json.dumps(
201                 best_coloring,
```

```
                    ensure_ascii=False,
                    indent=4))

        print(
            f"Хроматическое число жадная( раскраска): {max(greedy_coloring.values()) + 1}"
    )
        print(
            f"Хроматическое число оптимальная( раскраска): {max(best_coloring.values()) +
    1}")

    g.draw(greedy_coloring=greedy_coloring, best_coloring=best_coloring)


if __name__ == "__main__":
    main()
```

Листинг 1: Код программы на Python