Алгоритмы раскраски 3-дольных графов

Алексей Журавлев

11 декабря 2015 г.

1 Введение

Задачи поиска приближённого решения различных NP-трудных задач решаются людьми уже несколько десятилетий. Примером такой задачи, в которой человечество слабо преуспело за последние 30 лет является задача о правильной раскраске графа в оптимальное число цветов. В этой статье описаны алгоритмы Виджерсона и Бергера-Ромпеля раскраски 3-раскрашиваемых графов в $O(\sqrt{n})$ и $O(\sqrt{\frac{n}{log(n)}})$ цветов соответственно, где n - число вершин в графе. Будет доказана корректность работы алгоритмов с точки зрения числа цветов и полиномиальность времени работы. Также будет приведена конкретная реализация алгоритмов и примеры их запусков на случайных графах.

2 Немного истории

Хронология появления полиномиальных алгоритмов раскраски 3-раскрашиваемого графа.

1. Виджерсон в 1983 году впервые публикует статью, в которой описан алгоритм покраски в $O(\sqrt{n})$ цветов.
2. Бергер, Ромпель, 1990 — улучшение, раскраска в $O(\sqrt{\frac{n}{log(n)}})$ цветов.
3. Блум, 1992 — раскраска в $O(n^{\frac{3}{8}})$ цветов.
4. Каргер, Мотвани, Судан, 1994 — раскраска в $O(n^{\frac{1}{4}})$ цветов.
5. Кламтак, 2007 — раскраска в $O(n^{0.2072})$ цветов.

Стоит заметить, что нижние оценки для NP-трудности этой задачи всё ещё далеки от этих результатов

1. Карп в 1972 в своей книге публикует доказательство NP-трудности поиска оптимальной 3-раскраски для 3-дольного графа.
2. Канна, Линиал, Сафра, 1993 — доказательство NP-трудности поиска раскраски 3-дольного графа в 4 цвета.

3 Алгоритм Виджерсона

Пусть дан граф $G = (V, E), |V| = n$, про который известно, что его можно раскрасить в 3 цвета. Обозначим $N(v) = \{u : (u, v) \in E\}$ - множество соседей вершины v в графе G

Алгоритм Виджерсона основан на простой идее, которая состоит в следующем: если граф, красится в 3 цвета, то для любой вершины v: $N(v)$ можно правильно покрасить в 2 цвета. Это так, потому что в правильной 3-раскраске у $N(v)$ цвета вершин отличны от цвета вершины v и, следовательно, принимают только 2 значения. Поиск же правильной 2-раскраски делается за полиномиальное время тривиальным поиском в глубину.

Другая идея состоит в том, что граф, в котором степень каждой вершины не превосходит d, можно быстро и правильно покрасить в $d + 1$ цвет.

Это действительно так: можно просто просматривать вершины по очереди, и красить каждую вершину в минимально возможный цвет с учётом уже покрашенных. Т. к. степень каждой вершины не превосходит d, то номер выбранного цвета не может быть больше, чем $d+1$.

Тогда алгоритм будет состоять в следующем:

1. Найдём вершину v степени $\geq \sqrt{n}$. Рассмотрим множество её соседей $N(v)$. Если такой вершины v не нашлось, переходим к шагу (3).

2. Покрасим $N(v)$ правильно в 2 цвета и удалим из графа вместе со смежными рёбрами. Данные 2 цвета больше не будем использовать при дальнейшей покраске других вершин. Вернёмся на шаг (1).

3. Последовательно просмотрим все не покрашенные вершины и покрасим их в минимально возможный цвет с учётом уже покрашенных.

Раскраска графа правильная по построению алгоритма. Докажем оценку на число цветов, в которые будет покрашен граф.

Утв. *Алгоритм Виджерсона красит 3-раскрашиваемый граф в $O(\sqrt{n})$ цветов.*

◀ Заметим, что шаг (2) не может быть выполнен больше чем $\sqrt{n}$ раз, т.к. каждый раз, он удаляет из графа хотя бы $\sqrt{n}$, вершин, а суммарно в графе n вершин. Каждый раз шаг (2) использует 2 новых цвета. Итого на шаге (2) будет использовано суммарно $\leq 2\sqrt{n}$ цветов. На шаге (3) не может быть использовано больше $\sqrt{n}+1$ цветов, т.к. степень каждой вершины на шаге (3) строго меньше $\sqrt{n}$. Итого суммарно использовано $\leq \sqrt{n}+1+2\sqrt{n} = 3\sqrt{n}+1 = O(\sqrt{n})$ цветов. ■

Покажем полиномиальность алгоритма.

Утв. *Алгоритм Виджерсона строит покраску 3-раскрашиваемого графа за время $O(n^2)$*

◀ При покраске множества вершин в 2 цвета каждый раз запускается поиск в глубину. При этом, если вершина покрашена в этом поиске в глубину, то вершина и все инцидентные рёбра больше не будут посещены никаким поиском в глубину, т. к. исключаются из рассмотрения после покраски. Значит суммарное время работы всех поисков в глубину на шаге (2): $O(|V|+|E|) = O(n^2)$. На шаге (1) можно найти вершину максимальной степени за $O(n)$, поэтому суммарное время работы шага (1): $O(n^{\frac{3}{2}})$. Выбор цвета на шаге (3) выполняется за $O(\sqrt{n})$, вершин на шаге (3) нужно обработать не больше $O(n)$. Суммарное время на шаге (3) не больше, чем $O(n^{\frac{3}{2}})$. Итоговое время работы $O(n^2)$ ■

4 Реализация алгоритма Виджерсона

Приведём реализацию алгоритма Виджерсона на языке Python. Версия интепретатора 3.5.0. Граф договоримся хранить в виде списков смежности. Вершины имеют номера от 0 до $n-1$.

Для начала опишем основные вспомогательные функции для реализации алгоритма.

Первая из них $color\_subset(graph, important\_verteces, colors, min\_color)$ — принимает на вход граф, список вершин, которые нужно покрасить в 2 цвета, список соответствующий текущей покраске вершин ($None$ - соответствует не покрашенной вершине), а также номер последнего не использованного цвета. Покраска производится тривиально: поиском в глубину. Для экономии памяти на стеке, функция поиска в глубину внесена внутрь основной как замыкание. Возвращает функция значение $min\_color+2$, как новый минимальный не использованный цвет. Если в процессе обхода выяснилось, что окрестность не является 2-раскрашиваемой, то возвращается $None$.

Вторая функция $recalculate\_degrees(graph, important\_verteces, degrees)$ — пересчитывает степени всех смежных вершин после выполнения предыдущей процедуры, в соответствии с тем, что покрашенные вершины считаются удалёнными из графа. На вход подаётся граф, тот же список уже покрашенных вершин и список текущих степеней.

Третья функция $calculate\_greedy\_rest(graph, colors)$ — красит множество вершин графа, которые ещё не покрашены, жадным алгоритмом (в минимально возможный цвет).

help_functions.py

```python
def color_subset(graph, important_verteces, colors, min_color):
    vertices_set = set(important_verteces)
    def dfs(v, shift):
        colors[v] = min_color + shift
        for incident in graph[v]:
            if incident in vertices_set:
                if colors[incident] is None:
                    if not dfs(incident, 1 - shift):
                        return False
                elif colors[incident] == colors[v]:
                    return False
        return True
    for vertex in important_verteces:
        if colors[vertex] is None:
            if not dfs(vertex, 0):
                return None
    return min_color + 2


def recalculate_degrees(graph, important_verteces, degrees):
    for vertex in important_verteces:
        for incident in graph[vertex]:
            degrees[incident] -= 1
    for vertex in important_verteces:
        degrees[vertex] = 0
    return


def calculate_greedy_rest(graph, colors):
    n = len(graph)
    for vertex in range(n):
        if colors[vertex] is None:
            neighbour_colors = set(range(n))
            for neighbour in graph[vertex]:
                if colors[neighbour] is not None and colors[neighbour] in neighbour_colors:
                    neighbour_colors.remove(colors[neighbour])
            colors[vertex] = min(neighbour_colors)
```

Теперь приведём реализацию основного алгоритма. Алгоритм оформлен в виде класса, в конструктор которому подаётся граф, после чего там же происходит раскраска графа алгоритмом Виджерсона. Реализация в точности повторяет словесное описание из предыдущего пункта статьи: берётся вершина максимальной степени, рассматривается множество её соседей и красится в 2 цвета, после чего пересчитываются степени всех вершин. Когда максимальная степень вершины меньше, чем $\sqrt{n}$, переходим к жадной раскраске оставшейся части графа.

Получить раскраску графа пользователь может методом класса $get\_coloring()$.

widgerson.py

```
from math import sqrt
from help_functions import color_subset, recalculate_degrees, calculate_greedy_rest


class Widgerson:

    def __init__(self, graph):
        n = len(graph)
        degrees = [len(incident_list) for incident_list in graph]
        border = sqrt(n)
        self.colors = [None] * n
        min_color = 0
        while True:
            max_degree = max(degrees)
            if max_degree < border:
                break
            current_vertex = degrees.index(max_degree)
            min_color = color_subset(graph, graph[current_vertex], self.colors, min_color)
            recalculate_degrees(graph, set(graph[current_vertex]), degrees)
        calculate_greedy_rest(graph, self.colors)


    def get_coloring(self):
        return self.colors
```

5 Алгоритм Бергера-Ромпеля

Является во всех смыслах улучшением предыдущего алгоритма, обобщая при этом всё ту же общую идею.

В то время, как Виджерсон пользуется тем, что окрестность любой вершины является двудольной, Бергер и Ромпель пользуются тем, что окрестность любого независимого множества в графе G, которое одноцветно в некоторой правильной 3-раскраске, является двудольной из тех же рассуждений. Вопрос лишь в том, как такое множество искать.

Для $S \subset V$ будем обозначать $N(S) = \bigcup\limits_{u \in S} N(u)$.

Алгоритм будет состоять в следующем:

1. Положим $S = \emptyset$

2. Если существует вершина $u \in V$, которая добавляет хотя бы $\sqrt{\frac{n}{\log n}}$ соседних вершин к соседним вершинам множества S, то есть формально $|N(u) \setminus N(S)| \geq \sqrt{\frac{n}{\log n}}$, то добавим u в множество S. Если такой вершины нет, переходим к шагу 5.

3. Если $|S| \geq 3 \log n$, переходим к шагу 4, иначе возвращаемся на шаг 2.

4. Для каждого $C \subset S : |C| = \log n$, являющегося независимым множеством, запустим алгоритм покраски в 2 цвета множества $N(C)$, пока одно из них не покрасится правильно в 2 цвета. После этого удаляем покрашенные вершины из графа и не используем эти 2 цвета при дальнейшей покраске. Возвращаемся к шагу 1.

5. Для всех вершин, которые ещё остались в множестве S, рассмотрим последовательно их окрестности и раскрасим эти окрестности в 2 цвета, удаляя их после этого из графа.

6. Все оставшиеся вершины просматриваем последовательно и красим жадно: в минимально возможный цвет с учётом текущей раскраски.

Только пункт 4 этого алгоритма ставит под вопрос его корректность. Покажем, что он правомерен.

Утв. *В пункте 4 алгоритма найдётся независимое множество C, такое что окрестность $N(C)$ можно правильно раскрасить в 2 цвета.*

◀ Заметим, что $|S| \geq 3\log n$. При этом граф G — 3-раскрашиваемый, а значит из принципа Дирихле в S существует подмножество C, такое что $|C| \geq \log n$ и в правильной раскраске покрашено в один цвет. Тогда $N(C)$ будет в этой раскраске покрашено в 2 цвета. ■

Итак, алгоритм строит правильную раскраску. Докажем оценку на число цветов в ней.

Утв. *Алгоритм Бергера-Ромпеля красит 3-раскрашиваемый граф в* $O(\sqrt{\frac{n}{\log n}})$ *цветов*

◀ Покраска вершин происходит на 4, 5, 6 шагах алгоритма.

Заметим, что на 4 шаге алгоритма, после покраски из графа удаляется $\geq \log n\sqrt{\frac{n}{\log n}} = \sqrt{n\log n}$ вершин, т.к. по построению множества на шаге 2 каждая вершина добавляет $\sqrt{\frac{n}{\log n}}$ своих соседей, а $|C| \geq |\log n|$. Так как всего в графе G n вершин, то шаг 4 не может быть выполнен больше, чем $\sqrt{\frac{n}{\log n}}$ раз. Каждый раз используется 2 цвета, значит суммарно на шаге 4 используется $\leq 2\sqrt{\frac{n}{\log n}}$ цветов.

На шаге 5 нужно последовательно покрасить в 2 цвета окрестности менее чем $3\log n$ вершин. Для этого требуется менее чем $6\log n$ цветов.

На шаге 6 у каждой вершины степень меньше, чем $\sqrt{\frac{n}{\log n}}$, т.к. в противном случае вершина была бы добавлена в множество S. Значит жадный алгоритм покрасит все оставшиеся вершины в $\leq \sqrt{\frac{n}{\log n}} + 1$ цветов.

Итого, общее число цветов $\leq 2\sqrt{\frac{n}{\log n}} + 6\log n + \sqrt{\frac{n}{\log n}} + 1 = O(\sqrt{\frac{n}{\log n}})$ ■

Остаётся показать полиномиальность времени работы алгоритма.

Утв. *Алгоритм Бергера-Ромпеля красит 3-раскрашиваемый граф за время* $O(n^6)$.

◀ Посчитаем число операций на шаге 4. Шаг 4 выполняется $\leq \sqrt{\frac{n}{\log n}}$ раз. Каждый раз запускается серия из $(3\log n)^{\log n} \leq n^3$ поисков в глубину, каждый из которых который работает $O(n^2)$. Итого $O(n^6)$ операций на 4 шаге.

На шаге 2 нужно для каждой вершины проверить, как её добавление влияет на множество S. Это можно тривиально сделать за $O(n^2)$. Суммарно шаг 2 выполняется $\leq 3\log n\sqrt{\frac{n}{\log n}}$ раз. Итого $O(n^3)$ операций на этом шаге.

На шаге 5 просматривается не более, чем $3\log n$ окрестностей, каждая из которых красится за время $O(n^2)$. Итого $O(n^3)$ операций на этом шаге.

На шаге 6 линейным проходом просматриваются все вершины и каждый раз выбирается наименьший цвет. $O(n^2)$ операций на этом шаге.

Итак, получили общее время работы $O(n^6)$. ■

Заметим, что оценка получилась достаточно грубая, и в реальности можно добиться лучшей, но для доказательства полиномиальности этого достаточно.

6 Реализация алгоритма Бергера-Ромпеля

Опишем для начала ещё одну вспомогательную функцию.

get_all_neighbours(graph, important_vertices, colors) — возвращает множество всех соседей для данного множества вершин, при этом только те вершины, которые ещё не покрашены (не удалены с точки зрения алгоритма).

help_functions.py

```python
def get_all_neighbours(graph, important_vertices, colors):
    neighbours = set()
    for vertex in important_vertices:
        for incident in graph[vertex]:
            if colors[incident] is None and incident not in important_vertices:
                neighbours.add(incident)
    return neighbours
```

Теперь можно описать основной алгоритм, который оформлен в виде класса с интерфейсом, аналогичным интерфейсу для алгоритма Виджерсона.

berger_rompel.py

```python
from math import log2, sqrt
from itertools import combinations
from help_functions import get_all_neighbours, color_subset, calculate_greedy_rest


class BergerRompel:

    def __init__(self, graph):
        n = len(graph)
        self.colors = [None] * n
        logn = round(log2(n) + 0.5)
        border = round(sqrt(n / logn) + 0.5)
        min_color = 0
        while True:
            current_size = 0
            current_s = set()
            current_neighbours = set()
            for vertex in range(n):
                count = 0
                local_set = set()
                for incident in graph[vertex]:
                    if self.colors[incident] is None and incident not in current_neighbours\
                            and incident not in current_s:
                        local_set.add(incident)
                        count += 1
                if count >= border:
                    current_neighbours.update(local_set)
                    current_s.add(vertex)
                    current_size += 1
                if current_size >= 3 * logn:
                    for combination in combinations(current_s, logn):
                        neighbours = get_all_neighbours(graph, combination, self.colors)
                        result = color_subset(graph, neighbours, self.colors, min_color)
                        if result is None:
                            for neighbour in neighbours:
                                self.colors[neighbour] = None
                        else:
                            min_color = result
                            break
                    break
            if current_size < 3 * logn:
                for vertex in current_s:
                    neighbours = get_all_neighbours(graph, [vertex], self.colors)
                    if len(neighbours) > 0:
                        min_color = color_subset(graph, neighbours, self.colors, min_color)
                break
        calculate_greedy_rest(graph, self.colors)

    def get_coloring(self):
        return self.colors
```

Реализация дословно повторяет описание алгоритма.

7 Тестовые запуски алгоритмов

Для проверки правильности работы реализованных алгоритмов нужно научиться проверять правильность возвращаемой раскраски, а так же генерировать случайные 3-раскрашиваемые графы.

Определим функцию проверки правильности раскраски.

tests.py

```python
def check_coloring_correct(graph, coloring):
    n = len(graph)
    flag = True
    for vertex in range(n):
        for incident in graph[vertex]:
            if coloring[vertex] == coloring[incident]:
                flag = False
    return flag
```

Также определим функцию генерации случайного 3-дольного графа с размерами долей n, m, l, и вероятностью появления ребра p.

tests.py

```python
import random

def bernoulli(p):
    return random.uniform(0, 1) <= p


def generate_random_graph(n, m, l, p):
    total = n + m + l
    graph = []
    for i in range(total):
        graph.append([])
    for i in range(n):
        for j in range(m):
            if bernoulli(p):
                graph[i].append(n + j)
                graph[n + j].append(i)
        for k in range(l):
            if bernoulli(p):
                graph[i].append(n + m + k)
                graph[n + m + k].append(i)
    for j in range(m):
        for k in range(l):
            if bernoulli(p):
                graph[n + j].append(n + m + k)
                graph[n + m + k].append(n + j)
    return graph
```

Тестирование проведём на маленьких графах: треугольник, 2 несвязанных ребра. Получим корректные раскраски в обоих случаях.

Далее запустим оба алгоритма, на максимально допустимом с точки зрения времени числе вершин при различных значениях p. Получим следующие результаты.

Виджерсон			Бергер-Ромпель		
n	p	число цветов	n	p	число цветов
1000	0.5	6	1000	0.5	21
1000	0.05	28	1000	0.05	62
1000	0.2	12	1000	0.1	53
1500	0.1	22	-	-	-
2000	0.05	28	-	-	-
3000	0.05	33	-	-	-

Заметим, что число цветов не превышает заявленных оценок, однако на реальных практических примерах, на которых реализации удаётся запустить, алгоритм Виджерсона работает эффективнее алгоритма Бергера-Ромпеля.

Наиболее эффективно алгоритм Виджерсона работает на графах с большой плотностью рёбер.

Алгоритм Бергера-Ромпеля в данных примерах работает хуже из-за $6\log n$ в оценке на число цветов, которое в асимптотике уходит в $O(\sqrt{\frac{n}{\log n}})$, однако при $n = 1000$ несёт значительный вклад.

Тем самым алгоритм Виджерсона показывает себя достаточно быстрым, простым и эффективным на практике, по сравнению с алгоритмом Бергера-Ромпеля.

8 Итоги

В рамках этого обзора удалось рассмотреть только простейшие алгоритмы раскраски 3-дольных графов из множества алгоритмов, рассмотренных в начале этой статьи. Тем не менее, можно отметить, что простой и эффективный с точки зрения времени алгоритм Виджерсона на реальных примерах работает лучше, чем более сложные алгоритмы с асимптотически-лучшими оценками. Однако асимптотическую теорию нельзя не брать в расчёт, учитывая тенденции увеличения вычислительных мощностей с каждым годом.

Список литературы

[1] Bonnie Berger, John Rompel, A Better Perfomance Guarantee for Approximate Graph Coloring, Algoritmica, 1990

[2] Widgerson A., Improving the perfomance guarantee of approximate graph coloring, J. ACM, 30(4): 729-735, 1983

[3] Ryan O'Donnell, Coloring 3-Colorable Graphs using SDP, CMU Lecture, Spring 2008