МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«КУБАНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»**

**(ФГБОУ ВО «КубГУ»)**

**Факультет компьютерных технологий и прикладной математики**

**Кафедра вычислительных технологий**

**КУРСОВАЯ РАБОТА**

**АЛГОРИТМ РОЕВОГО ИНТЕЛЛЕКТА ДЛЯ РЕШЕНИЯ ЗАДАЧИ РАСКРАСКИ ГРАФА**

Работу выполнил \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_А. А. Алексиади

(подпись)

Направление 02.03.02 — «Фундаментальные информатика и информационные технологии»

Направленность (профиль) «Вычислительные технологии»

Научный руководитель,

канд. техн. наук, доц.\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Е.Е.Полупанова

 (подпись)

Нормоконтролер,

канд. техн. наук, доц.\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Е.Е.Полупанова

 (подпись)

**СОДЕРЖАНИЕ**

[Введение 3](#_Toc532555751)

[1 Постановка задачи раскраски графа 5](#_Toc532555752)

[2 Принцип работы муравьиного алгоритма 7](#_Toc532555753)

[2.1 Общий принцип работы 7](#_Toc532555754)

[2.2 Адаптация муравьиного интеллекта для задачи раскраски графа 8](#_Toc532555755)

[3 Реализация приложения 11](#_Toc532555756)

[3.1 Основные сведения о программе 11](#_Toc532555757)

[3.2 Особенности реализации 11](#_Toc532555758)

[3.3 Исследование параметров алгоритмов для получения лучшего параметра эффективности по времени 12](#_Toc532555759)

[4 Тестирование алгоритма 16](#_Toc532555760)

[Заключение 18](#_Toc532555761)

[Список использованных источников 19](#_Toc532555762)

[Приложение А. Основная программа 20](#_Toc532555763)

[Приложение Б. Алгоритм antColoring 22](#_Toc532555764)

# **Введение**

Раскраска графов находит применение не только в теоретических задачах, но и во многих практических областях, например, в решении задачи планирования или в технологии создания цифровых водяных знаков. Необходимость решения этой задачи на графах большого размера подталкивает к разработке алгоритмов, способных дать приемлемое (близкое к точному) решение за малое время работы.

В последние годы интенсивно развиваются алгоритмы поисковой оптимизации, которые называют поведенческими, интеллектуальными, метаэвристическими, вдохновленными (инспирированными) природой, роевыми, многоагентными, популяционными и т.д. Эффективность таких алгоритмов соизмерима, а часто превосходит эффективность ставших уже классическими эволюционных алгоритмов, среди которых наиболее известен генетический алгоритм. С помощью популяционных алгоритмов успешно решаются сложные оптимизационные задачи, например, задачи автоматизированного проектирования, синтеза сложных химических соединений, оптимального управления динамическими системами.

Сегодня интенсивно распространяется направление Swarm intelligence [1]—роевой интеллект, которое включает в себя:

1. метод роя частиц—метод численной оптимизации;
2. муравьиный алгоритм—алгоритм оптимизации муравьиной колонии;
3. пчелиный алгоритм—алгоритм, имитирующий поведение кормовых медоносных пчёл;
4. искусственная иммунная система—адаптивная вычислительная система, основанная на теоретической иммунологии;
5. алгоритм гравитационного поиска—алгоритм поиска, основанный на законе всемирного тяготения;
6. алгоритм капель воды—алгоритм оптимизации, использующий методы естественных рек и как они находят почти оптимальные пути к месту назначения;
7. алгоритм кукушки—алгоритм, основанный на гнездовом паразитизме некоторых видов кукушек.

В данной работе будет рассмотрен только алгоритм муравьиного интеллекта для решения задачи раскраски графа.

Таким образом, в рамках курсовой работы будут выполнены:

* постановка задачи раскраски графов,
* рассмотрение принципов работы муравьиного интеллекта в рамках решения задачи раскраски графа,
* реализация алгоритма на языке Python,
* анализ параметров, от которых зависит работа алгоритма,
* тестирование эффективности алгоритма по времени работы и по требуемой памяти.

# **Постановка задачи раскраски графа**

Проблема раскраски графа появилась в 1852 году, когда Францис Гутри пытался раскрасить карту Англии так, чтобы любые два смежные региона имели разные цвета. Он сформулировал Теорему о четырёх красках, в которой утверждал, что всякую расположенную на сфере карту можно раскрасить не более чем четырьмя разными цветами (красками) так, чтобы любые две области с общим участком границы были раскрашены в разные цвета (рисунок 1). Теорему о четырёх красках долгое время не удавалось ни доказать, ни опровергнуть. Позднее проблема четырёх красок переросла в проблему раскраски графов.



Рисунок 1 – Проблема четырёх красок

В теории графов раскраска графов является частным случаем разметки графов. При раскраске элементам графа ставятся в соответствие метки с учётом определенных ограничений; эти метки традиционно называются «цветами». В простейшем случае такой способ окраски вершин графа, при котором любым двум смежным вершинам соответствуют разные цвета, называется раскраской вершин. Аналогично раскраска рёбер присваивает цвет каждому ребру так, чтобы любые два смежных ребра имели разные цвета. Наконец, раскраска областей планарного графа назначает цвет каждой области, так, что каждые две области, имеющие общую границу, не могут иметь одинаковый цвет.

Раскраска вершин — главная задача раскраски графов, все остальные задачи в этой области могут быть сведены к ней. Например, раскраска рёбер графа—это раскраска верши него рёберного графа, а раскраска областей планарного графа—это раскраска вершин его двойственного графа.

Раскраска графов находит применение и во многих практических областях, а не только в теоретических задачах [2]. Помимо классических типов проблем, различные ограничения могут также быть наложены на граф, на способ присвоения цветов или на сами цвета. Этот метод, например, используется в популярной головоломке Судоку. В этой области всё ещё ведутся активные исследования.

Раскраска с использованием $k$ цветов называется $k$- раскраской. Наименьшее число цветов, необходимое для раскраски графа $G$называется его хроматическим числом $X(G)$. Для любого графа без петель выполняется неравенство $1\leq X(G)\leq n$, где n – количество вершин графа G.

# **Принцип работы муравьиного алгоритма**

## **Общий принцип работы**

Муравьиные алгоритмы исследуются европейскими учеными с середины 90-х годов [3]. Уже на сегодняшний день получены результаты для оптимизации таких сложных комбинаторных задач, как задача коммивояжера, задача раскраски графа, квадратичная задача о назначениях, задача оптимизации сетевых графиков и многие другие. Особенно эффективны муравьиные алгоритмы при динамической оптимизации процессов в распределенных нестационарных системах, например, трафиков в телекоммуникационных сетях.

Имитация самоорганизации муравьиной колонии составляет основу муравьиных алгоритмов оптимизации. Колония муравьев может рассматриваться как многоагентная система, в которой каждый агент (муравей) функционирует автономно по очень простым правилам. В противовес почти примитивному поведению агентов, поведение всей системы получается на удивление разумным.

Основу поведения муравьиной колонии составляет самоорганизация, обеспечивающая достижения общих целей колонии на основе низкоуровневого взаимодействия. Колония не имеет централизованного управления, и её особенностями являются обмен локальной информацией только между отдельными особями (прямой обмен – пища, визуальные и химические контакты) и наличие непрямого обмена, который и используется в муравьиных алгоритмах.

В реальном мире муравьи (первоначально) ходят в случайном порядке и по нахождении продовольствия возвращаются в свою колонию, прокладывая феромонами тропы. Если другие муравьи находят такие тропы, они, вероятнее всего, пойдут по ним. Вместо того, чтобы отслеживать цепочку, они укрепляют её при возвращении, если в конечном итоге находят источник питания. Со временем феромонная тропа начинает испаряться, тем самым уменьшая свою привлекательную силу. Чем больше времени требуется для прохождения пути до цели и обратно, тем сильнее испарится феромонная тропа. На коротком пути, для сравнения, прохождение будет более быстрым и как следствие, плотность феромонов остаётся высокой.

Испарение феромонов также имеет свойство избежания стремления к локально-оптимальному решению. Если бы феромоны не испарялись, то путь, выбранный первым, был бы самым привлекательным. В этом случае, исследования пространственных решений были бы ограниченными. Таким образом, когда один муравей находит (например, короткий) путь от колонии до источника пищи, другие муравьи, скорее всего пойдут по этому пути, и положительные отзывы в конечном итоге приводят всех муравьёв к одному, кратчайшему, пути.

##  **Адаптация муравьиного интеллекта для задачи раскраски графа**

Изначальная формулировка принципов муравьиного алгоритма предполагает нахождение кратчайшего пути вследствие привлечения все большего количества муравьев к кратчайшему пути с помощью феромонов.

Для решения задачи раскраски графа необходимо модифицировать поведение муравьиной колонии. Принцип общения с помощью фероменов преобразуем в общение с помощью цветов. У каждого муравья есть его собственный цвет, который будет, как и феромоны, будет привлекать муравьев того же цвета. Также у каждого муравья есть номер вершины, где он сейчас находится. Радиус «зрения» муравья, на котором он видят других муравьев и их цвета ограничивается списком смежных с текущей вершин.

Принцип работы колонии заключается в том, что муравьи перемещается от одной вершины к другой вершине графа, связанными ребрами. Предполагается, что муравьи могут достигать любой вершины из любой другой вершины, связанной с ней путём в графе.

Алгоритм создаёт $N^{2}$ муравьёв – $N$ муравьёв для каждой из $N$ вершин графа – начальную колонию. Каждому муравью присваивается случайный цвет из набора цветов $C= \left\{1, …,M\right\}$.

На каждой итерации происходят следующие изменения графа и перемещения муравьёв.

1. Для каждой вершины графа цвет определяется большинством муравьёв, находящихся в ней. Какого цвета муравьёв больше в вершине, такой цвет ей и присваивается.
2. Муравьи, которые остались в меньшинстве в вершине относительно присвоенного ей цвета, отправляются в соседние вершины. Предпочтительны вершины такого цвета, который совпадает с цветом муравья.
3. Каждая вершина привлекает к себе муравьёв своего цвета из смежных вершин. Конфликты решаются в пользу вершин с наибольшим числом муравьев, имеющих цвет данной вершины. Здесь проявляется принцип работы феромонов. Как было описано выше, муравьи привлекают других муравьев, имеющих тот же цвет. Если в одной вершине количество муравьев определенного цвета больше, то и их влияние будет больше.
4. Если вершина оказалась пустой (все муравьи, следуя правилам, описанным выше, покинули её), то для неё вновь генерируется $N$ муравьёв со случайными цветами.
5. Выполняется проверка корректности найденной раскраски графа: если ни одна пара смежных вершин не окрашена одинаковым цветом, то раскраска считается корректной и алгоритм завершает свою работу.

Интуитивно ожидается, что этот алгоритм будет сходиться к правильной раскраске графа, если количество цветов в выбранном наборе C больше хроматического числа графа, так как третий шаг гарантирует, что никакие две смежные вершины не будут иметь одинаковый цвет, а четвертый шаг гарантирует, что никакая из вершин графа не останется неокрашенной.

Если хроматическое число больше, чем количество цветов в наборе, то алгоритм не будет сходиться к стационарному состоянию, муравьи будут продолжать двигаться бесконечно, а их количество будет постоянно расти.

Чтобы избежать данной проблемы, добавим следующий механизм.

1. Начальный набор цветов C будет содержать количество цветов, равное нижней оценке хроматического числа по Геллеру [4], равной $\frac{n^{2}}{n^{2}-2m}$, где n – количество вершин в графе, а m – количество ребер.
2. Число возможных итераций (переходов муравьев и пересчета цветов) на каждом наборе цветов C ограничим константой maxiter, дабы избежать бесконечной работы алгоритма.
3. Если число итераций превышает параметр maxiter, то добавляем еще один цвет в набор цветов C, сгенерируем новую колонию муравьев, и алгоритм продолжит работу с новым набором цветов
4. В конце концов алгоритм либо достигнет такого состояния, когда раскраска, полученная муравьями, считается корректной – тогда алгоритм завершит свою работу, либо количество цветов в наборе будет равно числу вершин в графе. В этом случае вершинам назначаются попарно различные цвета из набора и алгоритм тоже будет считаться завершенным.

# **Реализация приложения**

## **Основные сведения о программе**

Для реализации программы используется язык программирования Python 3.7 и вспомогательные пакеты NetworkX 2.2 (для генерации случайных графов и отрисовки графов) и Matplotlib (для отрисовки графов).

В качестве основного интерфейса для ввода графа выступает файл inputGraph.txt. Пользователю предлагается ввести матрицу смежности графа, на основе которой потом выполняется алгоритм. Матрица смежности представляется собой n сток, в каждой из которых n цифр – 0 (если ребра нет) и 1 (если ребро есть) – разделенных пробелом.

В выходной файл output.txt записывается результат работы алгоритма –список цветов (целых чисел, кодирующих некоторые цвета), назначенных вершинам в порядке увеличения их порядковых номеров.

Также программа отрисовывает исходный граф и граф с найденной раскраской (цвета для отрисовки выбираются случайно) и сохраняет их в виде изображения с расширением png.

## **Особенности реализации**

Исходный код программы разделен на 2 файла:

1. main.py – основной файл, необходимый для взаимодействия с пользователем, содержит 3 функции:
	1. read\_from\_file() – функция, с помощью которой считывается матрица смежности из входного файла;
	2. random\_graph(n,d) – функция, генерирующая случайный граф с заданным числом вершин (входной параметр n) и заданной степенью каждой вершины (входной параметр d), при этом генерация выполняется на основе процедуры из пакета NetworkX;
	3. show\_graph(graph,colors) – процедура, которая рисует исходный граф и граф с правильной раскраской, используя найденную алгоритмом конфигурацию цветов (входной параметр colors);
2. antColoring – файл алгоритма:
	1. класс Ant с двумя параметрами: color – цвет муравья, pos – номер вершины, на которой находится муравей;
	2. create\_colony(n, colorCount) – функция, генерирующая начальную колонию муравьев, основывается на количестве вершин графа (входной параметр n) и количестве цветов в наборе (входной параметр colorCount);
	3. check\_coloring(graph,colors) – функция, проверяющая корректность текущей раскраски графа;
	4. coloring(g) – основная функция алгоритма, в которой выполняется муравьиный алгоритм, рассчитываются цвета для вершин графа.

## **Исследование параметров алгоритмов для получения лучшего параметра эффективности по времени**

В алгоритме присутствуют два параметра, корректировка которых может повлиять на его эффективность.

1. Параметр maxiter – количество итераций, которые будут выполнены на одном наборе цветов С. Очевидно, что слишком маленькое значение параметра может не позволить алгоритму сойтись к стационарному состоянию, будучи запущенным на графах большого размера. В случае, когда параметр имеет слишком большое значение, время выполнения программы будет велико, сходимость набора цветов к подходящему для раскраски графа будет слишком долгой.

Проведем тестирование алгоритма с разными значениями maxiter на графах разного размера. График эффективности алгоритма представлен на рисунках 2, 3 и 4.



Рисунок 2 – График зависимости времени работы от параметра maxiter на графах с 20 вершинами



Рисунок 3 – График зависимости времени работы от параметра maxiter на графах с 100 вершинами



Рисунок 4 – График зависимости времени работы от параметра maxiter на графах с 500 вершинами

 Выберем некое среднее значение, при котором алгоритм работает минимально возможное время на графах разного размера. Исходя из испытаний, таким значением является 80.

1. Вторым параметром является количество муравьев, генерируемых для каждой вершины в начальной колонии. Изначально этот параметр равен количеству вершин в графе. Однако есть вероятность, что его изменение может повлиять на эффективность алгоритма. График тестирования представлен на рисунке 5.



Рисунок 5 – Фрагмент графика зависимости времени работы от числа сгенерированных муравьев для одной вершины

Тестирование проводилось на графе, имеющим 500 вершин. На данном фрагменте графика видно, что лучшая скорость достигается при значении параметра приблизительно равном $\frac{3}{4}N$, так как от количества сгенерированных муравьев напрямую зависит время выполнения одной итерации. Как показывают тесты, для графов небольшого размера значение этого параметра не так сильно сказывается на времени работы и на точности алгоритма, следовательно найденное значение можно считать оптимальным.

# **Тестирование алгоритма**

После корректировки параметров алгоритма интересным является тестирование алгоритма с точки зрения скорости работы и с точки зрения точности раскраски графа. Под точностью будем понимать отклонение количества использованных цветов от хроматического числа графов.

Сравнение будем проводить с точным полнопереборным алгоритмом раскраски графа. Точность данного алгоритма будем считать стопроцентной, так как принцип его работы подразумевает восходящий перебор всех возможных раскрасок графа, пока не будет найдена корректная раскраска с минимальным возможным количеством использованных цветов.

В качестве контрольного графа будет граф Турана T(13,4) [5]. Он будет иметь 13 вершинами, разделенными на 4 подмножества с как можно более близким размером (рисунок 6). Вершины в таком графе соединены ребром, если они принадлежат разным подмножествам.



Рисунок 6 – Граф T(13,4)

Известно, что граф T(n,r) является r - раскрашиваемым [5]. Выполним над графом, представленном на рисунке 6, графом муравьиный алгоритм. Найденная раскраска корректна (следует из логики алгоритма) и содержит 4 цвета (рисунок 7). Заметим, что в других случаях и на других графах результат может быть хуже, так как алгоритм полагается на генератор случайных чисел при генерации популяций и при выборе муравьями вершины для перехода.



Рисунок 7 – Раскрашенный граф T(13,4)

Для сравнения скорости работы будем использовать полные графы, так как для муравьиного алгоритма полный граф является худшим случаем, при котором будет выполнено максимальное количество итераций. Результаты сравнения скорости работы представлены в таблице 1.

Таблица 1 – Результаты времени выполнения муравьиного и полнопереборного алгоритмов при раскраске полных графов

|  |  |
| --- | --- |
| Алгоритм | Время выполнения, сек. |
| K10 | K50 | K100 |
| Муравьиный | 0.000674 | 0.049864 | 0.311979 |
| Полнопереборный | 5.39576 | 194.7623 | 4861.7258 |

# **ЗАКЛЮЧЕНИЕ**

В ходе проведения курсовой работы была исследована модель муравьиного интеллекта. Были изучены общие принципы его работы, а также разработана его модификация, используемая для решения задачи раскраски графа.

В рамках исследования этого алгоритма была реализована программа с консольным интерфейсом, в которой можно искать решение задачи раскраски графа как на произвольном, задаваемом пользователем, графе, так и на случайно сгенерированном. Был проведен анализ параметров алгоритмов для того, чтобы найти оптимальные по результативности и времени значения параметров. Также было выполнено сравнительное тестирование муравьиного алгоритма с полнопереборным точным алгоритмом раскраски графа. По результатам тестирования можно заключить, что скорость работы муравьиного алгоритма значительно превосходит скорость полнопереборного аналога. Хотя в некоторых случаях из-за доли случайности возможен «пропуск» раскраски с использованием количества цветов равного хроматическому числу графа, точность алгоритма (относительно количества использованных цветов) все еще остается на высоком уровне. В большинстве случаев алгоритм выдает точную раскраску за счет восходящего перебора количества используемых цветов.

# **Список использованных источников**

1. Swarm Algorithms [Электронный ресурс]. – URL: http://www.cleveralgorithms.com/nature-inspired/swarm.html (дата обращения: 1.12.2018).
2. Tommy R. Jensen,  Bjarne Toft. Graph Coloring Problems / Tommy R. Jensen,  Bjarne Toft. – John Wiley & Sons – 1994. – 304с.
3. Marco Dorigo, Thomas Stützle. Ant Colony Optimization. / Marco Dorigo, Thomas Stützle. – A Bradford Book; First Edition, First Printing edition – 2004. – 264с.
4. Верхние и нижние оценки хроматического числа [Электронный ресурс]. – URL: <https://neerc.ifmo.ru/wiki/index.php?title> = Верхние\_и\_нижние\_оценки\_ хроматического\_числа (дата обращения: 1.12.2018).
5. Turán graphs [Электронный ресурс]. – URL: https://www.mscs.dal.ca/~janssen/4115/presentations/Thomas.pdf (дата обращения: 2.12.2018г.)
6. Python 3.7.1. [Электронный ресурс]. – URL: https://www.python.org/ (дата обращения: 1.12.2018г.).
7. NetworkX 2.2 [Электронный ресурс]. – URL: http://networkx.github.io/documentation/networkx-2.2/ (дата обращения: 1 декабря 2018).
8. MatPlotLib 3.0.2 [Электронный ресурс]. – URL: https://matplotlib.org (дата обращения: 2.12.2018).

# ПРИЛОЖЕНИЕ А

**Основная программа**

Файл main.py:

import networkx as nx
import time
import random
import matplotlib.pyplot as plt
from antColoring import coloring

# Считываем граф из текстового файла
def read\_from\_file():
 inputMatrix = open('input.txt', 'r')
 graph = []
 for row in inputMatrix:
 graph.append(row.replace('\n', '').split(' '))
 for i in range(len(graph)):
 for j in range(len(graph)):
 graph[i][j] = int(graph[i][j])
 return graph

# Генерируем случайный граф
def random\_graph(n,d):
 # graph = nx.fast\_gnp\_random\_graph(n, 0.3)
 graph = nx.random\_regular\_graph(d,n)
 m = nx.to\_numpy\_matrix(graph)
 graph = []
 for i in range(len(m)):
 row = []
 for j in range(len(m)):
 row.append(int(m[i,j]))
 graph.append(row)
 return graph

# Отрисовка графа без отмеченного паросочетания и с ним
def print\_graph(graph,colors):
 g = nx.DiGraph()
 edges = []
 for i in range(len(graph)):
 for j in range(len(graph)):
 if graph[i][j] == 1:
 edges.append((i,j))
 g.add\_edges\_from(edges)

 r = lambda: random.randint(0, 255)

 nodes = list(g.nodes.keys())
 match = [i for i in range(len(colors))]
 for i in range(len(colors)):
 match[i] = colors[nodes[i]]
 colors = match

# Получение случайных цветов
 for i in colors:
 if isinstance(i,int):
 color = '#{:02x}{:02x}{:02x}'.format(r(), r(), r())
 for j in range(len(colors)):
 if i == colors[j]:
 colors[j] = color

 pos = nx.spring\_layout(g)
 nx.draw\_networkx\_nodes(g, pos, cmap=plt.get\_cmap('jet'),
 node\_color='LightGray', node\_size=300)
 nx.draw\_networkx\_labels(g, pos)
 nx.draw\_networkx\_edges(g, pos, edgelist=g.edges, edge\_color='black', arrows=False)
 plt.show()

 pos = nx.spring\_layout(g)
 nx.draw\_networkx\_nodes(g, pos, cmap=plt.get\_cmap('jet'),
 node\_color=colors, node\_size=300)
 nx.draw\_networkx\_labels(g, pos)
 nx.draw\_networkx\_edges(g, pos, edgelist=g.edges, edge\_color='black', arrows=False)
 plt.show()

graph = random\_graph(7,4)
s1 = time.time()
colors = coloring(graph)
print\_graph(graph,colors)

# **ПРИЛОЖЕНИЕ Б**

**Алгоритм antColoring**

Файл antColoring.py:

from random import randint

maxiter = 80

class Ant:
 def \_\_init\_\_(self, color, pos):
 self.color = color
 self.pos = pos

def create\_colony(n,colorCount):
 colony = []
 for i in range(n):
 for j in range(n):
 ant = Ant(randint(0, colorCount), i)
 colony.append(ant)
 return colony

def check\_coloring(graph,colors):
 n = len(graph)
 flag = True
 for i in range(n):
 for j in range(i + 1, n):
 if graph[i][j] == 1 and colors[i] == colors[j]:
 flag = False
 return flag

def coloring(g):

 # находим количество вершин в графе
 n = len(g)

 # вычисляем количество ребер в графе,
 # так как граница количества цветов n^2/(n^2-2m)
 # также создаем список смежных вершин для каждой вершины
 numberEdges = 0
 spisok = {i: [] for i in range(n)}
 for i in range(n):
 for j in range(n):
 if g[i][j] == 1:
 if i > j:
 numberEdges += 1
 spisok[i].append(j)
 print(spisok)

 numberEdges = int(n \* n / (n \* n - 2 \* numberEdges))
 colors = [0]\*n

 while numberEdges != n:

 # создаем начальную колонию муравьев
 colony = create\_colony(n,numberEdges)

 for iter in range(maxiter):
 nodes = [{color: 0 for color in range(numberEdges + 1)} for j in range(n)]
 for ant in colony:
 nodes[ant.pos][ant.color] += 1
 for i in range(len(nodes)):
 maxInd = 0
 max = 0
 for j in nodes[i].keys():
 if nodes[i][j] > max:
 maxInd = j
 max = nodes[i][j]
 colors[i] = maxInd

 for ant in colony:
 if ant.color != colors[ant.pos]:
 flag = True
 for i in spisok[ant.pos]:
 if colors[i] == ant.color:
 ant.pos = i
 flag = False
 if flag:
 ant.pos = spisok[ant.pos][randint(0, len(spisok[ant.pos]) - 1)]

 for i in range(n):
 for j in range(i + 1, n):
 if g[i][j] == 1 and colors[i] == colors[j]:
 c1 = 0
 c2 = 0
 color = colors[i]
 for ant in colony:
 if ant.pos == i and ant.color == color:
 c1 += 1
 elif ant.pos == j and ant.color == color:
 c2 += 1
 if c1 > c2:
 for ant in colony:
 if ant.pos == j and ant.color == color:
 ant.pos = i
 else:
 for ant in colony:
 if ant.pos == i and ant.color == color:
 ant.pos = j

 buffer = set(node for node in range(n))
 for ant in colony:
 if ant.pos in buffer:
 buffer.remove(ant.pos)
 for node in buffer:
 for i in range(n):
 ant = Ant(randint(0, numberEdges), node)
 colony.append(ant)

 if check\_coloring(g,colors):
 return colors
 numberEdges += 1

return [color for color in range(numberEdges + 1)]