САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

КАФЕДРА КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ И СИСТЕМ

Грачев Дмитрий Алексеевич

Выпускная квалификационная работа бакалавра

Раскраска графов и задача построения расписания с ограничениями

Направление 010300

Фундаментальная информатика и информационные технологии

Научный руководитель

к. ф.-м. н., доцент

Погожев С. В.

Санкт-Петербург

2017

Содержание

Введение ... 3
Глава 1. Вершинная раскраска графа ... 5
Глава 2. Реализация алгоритмов раскраски ... 8
2.1. Алгоритм раскраски графа Красновой А. Ю. ... 8
2.2. Алгоритм раскраски графа ... 9
2.3. Сравнение алгоритмов ... 11
Глава 3. Задача о составлении расписания ... 14
3.1. Постановка задачи ... 14
3.2. Решение задачи ... 14
3.3. Пример работы программы ... 17
Заключение ... 20
Список литературы ... 21
Приложение ... 22
Приложение 1 ... 22
Приложение 2 ... 25

Введение

Теория графов активно развивается на протяжении последних нескольких десятилетий. Связано это с тем, что стремительно расширяется область приложений теории графов, увеличивается количество задач, которые могут быть решены посредством методов теории графов. Так, например, основополагающей для задач, связанных с навигацией и построением сетей, является задача нахождения кратчайшего пути на графе. Другой актуальной задачей является задача составления расписаний. Задача подразумевает составление абсолютно любого расписания: от школьного расписания до порядка проведения работ на предприятии. Ограничениями в данной задаче является невозможность проведения заданий (уроков, работ и т. п.) одновременно по ряду причин. Подобная задача возникает постоянно в различных ситуациях, поэтому для её решения необходим эффективный алгоритм.

Одной из трудных задач теории графов является задача отыскания хроматического числа графа, то есть минимального числа цветов, необходимых для раскраски вершин графа. Предложены различные алгоритмы решения данной задачи, однако поиск эффективного алгоритма продолжается. Раскраска вершин позволяет моделировать многие проблемы планирования. В частности, при помощи алгоритма раскраски графа может быть решена задача составления расписания.

В рамках данной работы рассматривается сведение задачи составления расписания с ограничениями к задаче о вершинной раскраске графа, осуществляется анализ алгоритма вершинной раскраски графа, предложенный Красновой А. Ю. [1] и предлагается альтернативный алгоритм раскраски графа.

План работы:

1. Задача раскраски графа и рассмотрение существующих алгоритмов её решения.

2. Реализация и анализ алгоритма, предложенного Красновой А. Ю. [1].
3. Разработка алгоритма решения задачи раскраски графа.
4. Сравнение результатов работы алгоритмов, реализованных в пунктах 2 и 3.
5. Сведение задачи составления расписания с ограничениями к задаче раскраски графа и применение разработанного алгоритма к её решению.

В первой главе работы приведены некоторые определения из теории графов, используемые в работе (граф, смежные вершины, матрица смежности), описывается задача раскраски графа, её варианты, а также подходы к нахождению неточного решения: последовательный перебор, жадные алгоритмы и алгоритмы с использованием битовых операций.

Вторая глава описывает схему и реализацию алгоритма, предложенного Красновой А. Ю., и разработанного в рамках данной работы алгоритма. В заключении главы представлено сравнение работы на коллекции DIMACS-файлов.

Третья глава посвящена задаче о составлении расписания. В начале главы приводится постановка задачи в её самом простом случае и вариант усложнения задачи. Далее в главе рассмотрен пошагово способ решения задачи: от исходных данных до результата. В конце главы приводится пример задания на тестовом расписании.

В заключении перечислены результаты работы и приводятся примеры разработок программных продуктов для составления расписания.

Глава 1. Вершинная раскраска графа

В данной главе будет рассмотрена задача вершинной раскраски обыкновенного графа. Перед тем как приступать к рассмотрению задач напомним определение графа, которое будет использовано далее, без углубления в теорию графов. Под «графом» в данной работе будет пониматься обыкновенный неориентированный граф (рис. 1.1). Стоит заметить, что большинство определений теории графов даются по-разному в различных источниках, но они просто демонстрируют разные подходы и в большинстве своём не противоречат друг другу. Определения можно найти в [2, 3, 4, 5].

- *Граф* – упорядоченная пара множеств (V, E), где V – непустое множество вершин, а E – множество неупорядоченных пар вида (v_i, v_j), называемых рёбрами, где v_i и v_j принадлежат множеству V.
- Вершины v_i и v_j графа $G = (V, E)$ называются *смежными*, если они соединены ребром, то есть если в E существует ребро (v_i, v_j).
- Одним из способов задания графа является *матрица смежности* – матрица A размером $[N \text{x} N]$, где N – количество вершин, в которой элемент $a_{ij} = 1$, если в графе вершины i и j смежны, и 0 в противном случае.

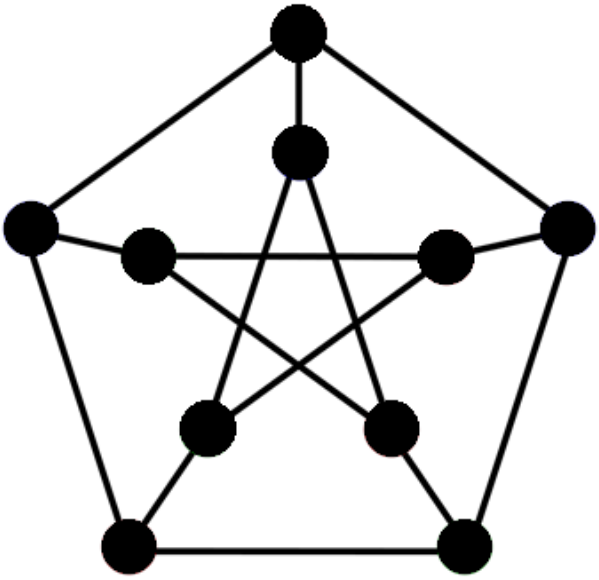


Рис. 1.1 Пример графа

Теперь рассмотрим подробнее задачу раскраски графа. Данная задача является одной из основных в теории графов, потому что входит в класс NP-полных и к ней сводятся многие другие задачи этой области. Раскраска графа может быть вершинной, рёберной и тотальной [6], однако все три варианта задачи можно свести друг к другу, поэтому рассмотрим только вершинную раскраску графа.

«Раскраской графа называется такое приписывание цветов его вершинам, что никакие две смежные вершины не получают одинакового цвета.» [5]

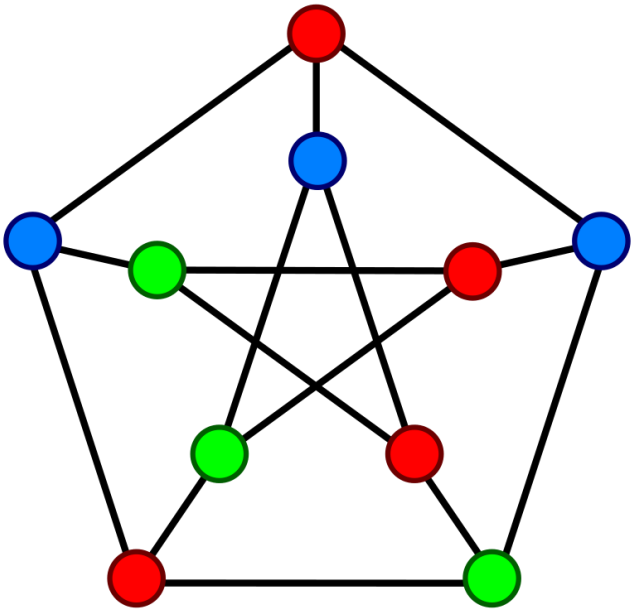


Рис. 1.2 Вариант раскраски графа на рис. 1.1

Задача же раскраски состоит в нахождении минимального количества цветов для раскраски графа. Такое решение далее будем называть *оптимальным*. На рис. 1.2 показана оптимальная раскраска графа, приведённого на рис. 1.1. Задача не имеет известного единого алгоритма для поиска решения за полиномиальное время. Существуют варианты таких решений только для определённых типов графов. Единственным подходом к нахождению оптимального решения является перебор вариантов. Однако предлагаются разные подходы и алгоритмы для поиска неточного, приближённого решения, которое найти гораздо проще оптимального. Приведем кратко принцип работы некоторых из них.

- *Алгоритмы последовательного перебора.* Такие алгоритмы последовательно отделяют группы цветов. На каждом этапе создаётся группа вершин, в которую пытаются включить нераспределённые вершины, пока все они не будут распределены по цветам.
- *Жадные алгоритмы.* Жадные алгоритмы упорядочивают вершины по какому-либо правилу (чаще всего встречаются примеры, где используют степени вершин) и последовательно присваивают им цвета, в которые не были окрашены смежные с ними вершины.
- *Алгоритм с использованием битовых операций.* Алгоритм похож на первый, но работает с матрицами, используя битовые операции. Такой подход позволяет сократить время работы в несколько раз [7].

Глава 2. Реализация алгоритмов раскраски

2.1. Алгоритм раскраски графа Красновой А. Ю.

В кандидатской диссертации Красновой А. Ю. «Матрицы инциденций и раскраски графа» предложен алгоритм раскраски обыкновенного графа, основанный на рассмотрении матрицы инциденций и последовательном выделении непустых подмножеств множества вершин графа, включающий процедуру перемешивания, позволяющую приблизить число красок к хроматическому числу.

Алгоритм представляет собой вариант алгоритма последовательного перебора, но с перемешиваниями и повторениями. Такое усовершенствование помогает уменьшить зависимость результата от начальной нумерации вершин. Алгоритм использует матрицу инциденций, а для сохранения исходной нумерации вершин к ней приписывается столбец I с номерами. Ниже приведен алгоритм [1].

«*ШАГ 1*. Пусть граф $G(V, X)$ задан матрицей смежности A. По ней строим матрицу инциденций B. Применяя к матрице алгоритм последовательного раскрашивания вершин, получаем раскраску в r цветов и матрицу перестановок P_0 такую, что $B_t = P_0B$.

ШАГ 2. Задаём число перемешиваний H.

ШАГ 3. Выполняем n итераций процедуры перемешивания. При этом полагаем, что $B_t^{(0)} = P_0B$ *и* $I_t^{(0)} = P_0I$.

Тогда k-тая итерация будет происходить так:

1. Строим матрицу инциденций $B_s^{(k)} = QB_t^{(k-1)}$.
2. Применяем к матрице $B_s^{(k)}$ алгоритм последовательного раскрашивания вершин, причём для построения «расширенной» матрицы инциденций $B_s^{(k)}$ приписываем слева к ней столбец $I_s^{(k)} = QI_t^{(k-1)}$. Получаем раскраску в r_k цветов и матрицу перестановок P_k такую, что $B_t^{(k)} = P_kB_s^{(k)}$. При этом $I_t^{(k)} = P_kI_s^{(k)}$.»

При реализации алгоритма было принято отказаться от использования

матрицы инциденций, поскольку матрица смежности графа позволяет уменьшить объём используемой оперативной памяти.

Реализация алгоритма приведена в приложении 1.

2.2. Алгоритм раскраски графа

Далее рассмотрим новый алгоритм раскраски графа, построенный как жадный алгоритм. В большинстве случаев, как уже отмечено ранее, жадные алгоритмы упорядочивают вершины по их степени (или относительной степени, то есть степени без учёта окрашенных вершин). В этом есть смысл, потому что окрашивание таких вершин приносит наибольшее количество информации в систему: с окраской таких вершин мы не можем окрасить большее количество цветов в цвет, выбранный для этой вершины. Для разработанного алгоритма было решено использовать степень вершин только как второстепенный фактор. Первичным же фактором для сравнения было выбрано количество цветов, в которые вершину окрасить уже нельзя. Во многих случаях такой подход формирует полный пул цветов, потому что рассматривает прежде всего вершины, для которых приходится добавлять цвет. Кроме того, зная, что алгоритм предполагается использовать для решения задачи составления расписания, будем упорядочивать не только вершины, но и цвета, чтобы сделать распределение вершин, а соответственно и нагрузки, более равномерным.

В качестве входных данных алгоритм принимает файлы стандартного для описания графов формата DIMACS [8], что позволяет использовать его как отдельную программу для раскраски графов. Файл считывается единожды, дальнейшая работа происходит с матрицей смежности.

В алгоритме можно выделить следующие этапы работы:

1. Считать информацию из файла и составить по нему матрицу смежности.
2. Инициализировать вектор цветов (изначально пустой), содержащий вектора вершин, окрашенных в соответствующий цвет. Сюда добавляется окрашенная вершина.

3. Сформировать вектор вершин. Для хранения информации о вершинах создан класс *Vertex*, содержащий номер вершины в графе, её степень и вектор цветов, в которые её нельзя окрасить.
4. Упорядочить вектор вершин по приоритетам. Приоритет вершины больше, если её нельзя окрасить в большее количество цветов; если количество цветов равно, то приоритетнее вершина с большей степенью.
5. Подобрать цвет для вершины, проверяя с наименее распространённых. Все доступные цвета хранятся в векторе, упорядоченном по количеству вершин, окрашенных в данный цвет. Проверяется вхождение рассматриваемого цвета в вектор цветов, в которые нельзя покрасить вершину.
    a. Если цвет найден
        i. Окрасить вершину
        ii. Пометить цвет как запрещённый для всех соседей. Для этого по соответствующей строке матрицы смежности находятся все соседние вершины и в вектор запрещённых для них цветов добавляется текущий, если его ещё нет.
        iii. Пересортировать вершины и цвета. Нет необходимости рассматривать соответствующие векторы целиком.
            1. После окрашивания вершины увеличилось количество вершин данного цвета, поэтому проверяется соседний с ним в векторе цветов и, пока необходимо, цвета меняются местами.
            2. Каждая соседняя вершина получает дополнительный запрещённый цвет – её приоритет увеличивается, поэтому проверяются соседние вершины в списке вершин и, пока необходимо, вершины меняются местами.
        iv. Исключить вершину. В матрице смежности обнуляются соответствующие строка и столбец, чтобы эта вершина не

рассматривалась как соседняя, и вершина удаляется из вектора вершин.

b. Если цвет не найден, добавить цвет и выполнить п.3.a.

6. Повторять п.3 пока не будут окрашены все вершины (пока вектор вершин не опустеет).

Полный исходный код алгоритма можно найти в приложении 2.

2.3. Сравнение алгоритмов

При сравнении алгоритмов в качестве показателей работы алгоритмов были выбраны количество цветов и время работы алгоритма. Естественно, что время работы алгоритма Красновой А. Ю. зависело от заданного количества перемешиваний, поэтому проведем анализ работы алгоритма без перемешиваний и с определёнными количествами перемешиваний, с которыми алгоритм выдаёт результат с меньшим количеством цветов. Так как время работы алгоритма сильно растёт с увеличением числа перемешиваний, а некоторые графы для примера довольно большие, то верхней границей установим 50 перемешиваний.

В качестве коллекции графов, на которых будет производиться сравнение возьмем набор графов с [9]. Будем использовать графы, представленные в файлах *.col*. Результаты сравнений приведены в таблице. В скобках после имени файла указано оптимальное количество цветов, если доказано. Жирным выделены лучшие показатели алгоритма (или алгоритмов), справившегося с задачей максимально эффективно.

Имя файла	Кол-во вершин	Кол-во рёбер	1		2		3	
			N	t	N	t	N	t
fpsol2.i.1.col	496	11654	65	89	**65**	**29**	65	1414
fpsol2.i.2.col	451	8691	30	132	**30**	**22**	30	1039
fpsol2.i.3.col	425	8688	30	132	**30**	**18**	30	851
inithx.i.1.col	864	18707	54	263	**54**	**201**	54	10343
inithx.i.2.col	645	13979	31	260	**31**	**69**	31	3587
inithx.i.3.col	621	13969	31	220	**31**	**62**	31	3060
latin_square_10.col	900	307350	**136**	**334**	213	230	146	11017
le450_15b.col (15)	450	8169	**17**	**38**	22	22	20	1035
le450_15c.col (15)	450	16680	**25**	**74**	30	22	29	1105

le450_15d.col (15)	450	16750	**26**	**82**	31	22	29	1095
le450_25a.col (25)	450	8260	**25**	**31**	28	22	26	1073
le450_25b.col (25)	450	8263	**25**	**32**	27	21	26	1033
le450_25c.col (25)	450	17343	**29**	**66**	37	21	34	1041
le450_25d.col (25)	450	17425	**29**	**62**	35	21	35	1040
le450_5a.col (5)	450	5714	**10**	**45**	14	21	12	1039
le450_5b.col (5)	450	5734	**10**	**45**	13	21	12	1036
le450_5c.col (5)	450	9803	**10**	**54**	17	21	14	1041
le450_5d.col (5)	450	9757	**12**	**58**	18	20	14	1032
mulsol.i.1.col	197	3925	49	13	**49**	**1**	49	74
mulsol.i.2.col	188	3885	31	18	**31**	**1**	31	64
mulsol.i.3.col	184	3916	31	19	**31**	**1**	31	62
mulsol.i.4.col	185	3946	31	19	**31**	**1**	31	61
mulsol.i.5.col	186	3973	31	19	**31**	**1**	31	62
school1.col	385	19095	**20**	**58**	42	12	38	625
school1_nsh.col	352	14612	**30**	**40**	39	9	36	473
zeroin.i.1.col	211	4100	49	18	**49**	**1**	49	96
zeroin.i.2.col	211	3541	30	25	**30**	**1**	30	96
zeroin.i.3.col	206	3540	30	13	**30**	**1**	30	89
anna.col	138	986	**11**	**2**	12	0	11	27
david.col	87	812	**11**	**1**	12	0	11	6
homer.col	561	3258	**13**	**20**	15	41	13	2199
huck.col	74	602	**11**	**0**	**11**	**0**	11	4
jean.col	80	508	**10**	**0**	**10**	**0**	10	5
games120.col	120	1276	**9**	**1**	**9**	**0**	9	16
miles1000.col	128	6432	**42**	**9**	44	0	43	20
miles1500.col	128	10396	73	29	76	0	**73**	**20**
miles250.col	128	774	**8**	**0**	9	0	8	20
miles500.col	128	2340	**20**	**1**	22	0	20	20
miles750.col	128	4226	**31**	**4**	34	0	31	20
queen10_10.col	100	2940	**14**	**3**	16	0	14	9
queen11_11.col	121	3960	**16**	**4**	17	0	15	17
queen12_12.col	144	5192	**16**	**6**	20	0	17	29
queen13_13.col	169	6656	**18**	**9**	21	0	18	47
queen14_14.col	196	8372	**19**	**12**	23	1	19	72
queen15_15.col	225	10360	**21**	**15**	25	2	21	111
queen16_16.col	256	12640	**22**	**21**	25	3	22	171
queen5_5.col	25	320	**5**	**0**	8	0	**5**	**0**
queen6_6.col	36	580	9	0	11	0	**8**	**0**
queen7_7.col	49	952	**10**	**0**	**10**	**0**	**10**	**1**
queen8_12.col	96	2736	**14**	**3**	15	0	14	8
queen8_8.col	64	1456	12	1	13	0	**11**	**2**
queen9_9.col	81	2112	**13**	**2**	16	0	13	5
myciel3.col (4)	11	20	**4**	**0**	**4**	**0**	**4**	**0**
myciel4.col (5)	23	71	**5**	**0**	**5**	**0**	**5**	**0**
myciel5.col (6)	47	236	**6**	**0**	**6**	**0**	**6**	**1**
myciel6.col (7)	95	755	7	3	**7**	**0**	7	8

myciel7.col (8)	191	2360	8	12	**8**	**0**	8	67

1 – собственный алгоритм, 2 – алгоритм Красновой без повторений, 3 – алгоритм Красновой с 50 повторениями, N – кол-во цветов, t – время в мс

Анализ результатов вычислений, приведенных в таблице позволяет сделать вывод, что предложенный жадный алгоритм показывает лучшие результаты в большинстве случаев. Особое внимание хотелось бы уделить файлам *school1.col* и *school1_nsh.col*. Эти описания соответствуют графам, полученными при рассмотрении задач о составлении расписания. Как видно по таблице, алгоритм Красновой А. Ю. справляется с ними гораздо хуже. Так, например, в случае с файлом *school1.col*, даже при 5000 повторениях количество цветов уменьшилось до 36, но при этом время работы составляло уже около 20 минут. С файлом *school1_nsh.col* ситуация похожая: при 5000 повторений количество цветов уменьшилось до 33, но время работы возросло до 15 минут. В обоих случаях количество цветов не достигло минимального полученного.

Глава 3. Задача о составлении расписания

3.1. Постановка задачи

Задача о составлении расписания с ограничениями подразумевает распределение занятий по времени таким образом, чтобы в одно и то же время не были назначены занятия у одной группы, одного преподавателя или в одном кабинете. Такая задача легко сводится к раскраске графов при условии, что известны все занятия для всех групп, которые надо провести, каждое занятие ведёт определённый преподаватель и известны кабинеты, в которых должны быть проведены занятия. В таком случае расписание может быть представлено графом, в котором вершины соответствуют занятиям, а рёбрами связаны вершины, соответствующие занятиям у одной группы, в одном кабинете или с одним преподавателем. После раскраски такого графа расписание сформируется по принципу «занятия, которые соответствуют вершинам одного цвета могут быть проведены одновременно».

Гораздо более сложный вариант задачи подразумевает, что для каждого занятия установлен тип кабинета, а всего кабинетов каждого типа ограниченное количество. Так, например, не получится провести 10 занятий в компьютерных классах, если их всего 5. Сложность задачи в такой постановке заключается в том, что подобные ограничения нельзя отобразить связями в графе. Для таких задач требуется дополнительная обработка либо в ходе решения, либо после получения раскраски графа.

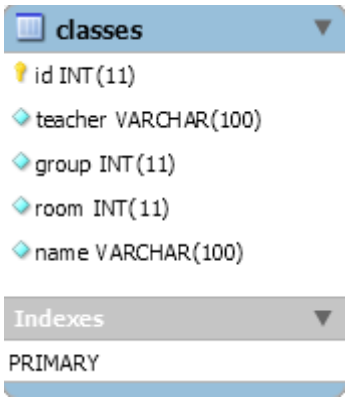
В данной работе далее будет подробно рассмотрен способ решения первого варианта задачи, а также рассмотрены усовершенствования задачи и решения, в том числе и описанный выше вариант.

3.2. Решение задачи

Как уже отмечалось ранее, для получения решения задача будет сведена к задаче раскраски графов, однако для прикладного решения необходимо гораздо больше этапов. Рассмотрим пошагово систему для решения задачи о

составлении расписания.

Первый шаг – предоставление исходных данных. Предлагаемое решение использует базу данных для хранения и добавления информации о занятиях, которые необходимо провести. В самом простом случае достаточно одной таблицы, содержащей занятия (рис. 3.1). Для удобной работы даже для пользователя, не знакомого с синтаксисом работы с базами данных, достаточно простого интерфейса с возможностью добавления и просмотра содержимого.



id – идентификатор занятия

teacher – преподаватель

group – группа, в которой проводится занятие

room – номер кабинета

name – название предмета

Рис. 3.1 Пример таблицы в MySQL

Второй шаг – преобразование исходных данных в граф. Для получения структуры графа потребуется несколько запросов:

- *Запрос всего содержимого*. На данном этапе становится известным количество занятий, преподавателей, аудиторий и формируются отношения между парами и вершинами (самый простой вариант – по id).
- *Запросы по ограничениям*. Необходимо выделить ограничения, накладываемые расписанием. Так по каждому типу ограничений проводится запрос. Например, «*найти все занятия, которые ведёт преподаватель X*». Каждая пара занятий, полученных по таким запросам будет отображаться ребром в формируемом графе.

После проведения всех запросов будет окончательно сформирован граф, соответствующий рассматриваемому расписанию.

Третий шаг – раскраска полученного на предыдущем шаге графа. Для

этого применяем алгоритм раскраски, описанный ранее. На выходе получаем распределение вершин по группам (цветам).

Четвёртый шаг – заключительный, преобразование полученного ответа к результатам исходной задачи. Здесь необходимо каждой вершине сопоставить занятие, пользуясь сформированными на втором шаге отношениями. Занятия, соответствующие вершинам первого цвета, проводятся первыми, далее проводятся занятия, соответствующие вершинам второго цвета, и так далее.

В качестве усложнений в задаче может предусматриваться не конкретный кабинет, а его тип (компьютерный класс, лекционная аудитория и т.п.). В таком случае в базе данных потребуется таблица с кабинетами и их типами, на первом шаге необходимо будет получить количество кабинетов каждого типа, а в процессе раскраски окрашивать вершины в определённый цвет только тогда, когда не превышен максимум классов данного типа.

Кроме того, можно учитывать пожелания преподавателей: в какое время они хотят проводить свои занятия. Для этого в базе данных потребуется дополнительно таблица с преподавателями и их предпочтениями на каждое время в расписание. Учесть такие пожелания можно в самом конце при распределении групп цветов по местам в расписании. Для каждой группы можно получить наиболее предпочтительное место в расписании, суммируя пожелания преподавателей, ведущих занятия в данной группе.

Пример сложной структуры в базе данных представлен на рис. 3.2.

- *в room_resrt записан требуемый тип кабинета*
- *в таблице teachers под номерами желание преподавателей проводить занятие в это время (в примере 1 – 30: 6 дней по 5 занятий), отмечаемое 1 (подходит) или -1 (не подходит).*
- *в таблице rooms информация о всех имеющихся кабинетах: номер и тип.*

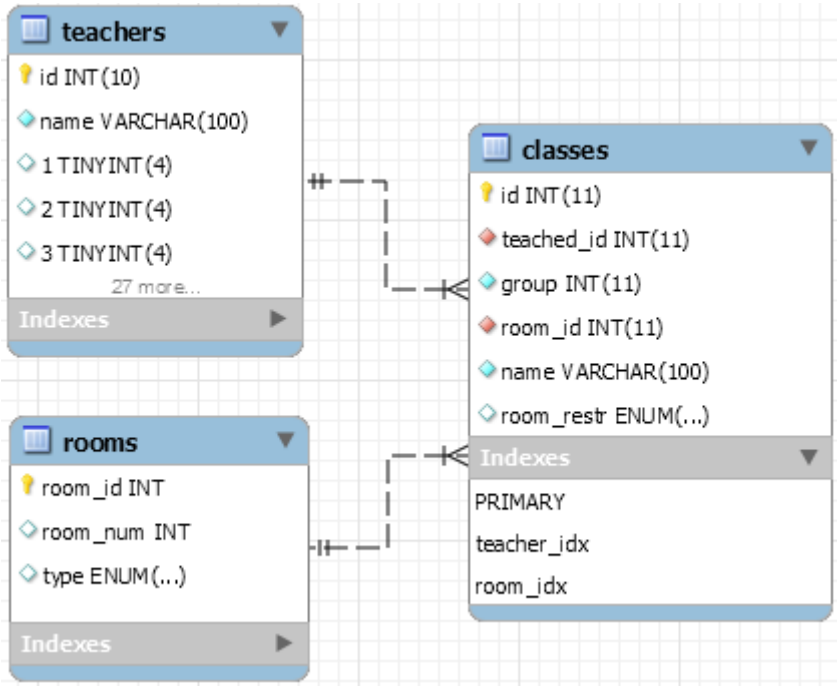


Рис. 3.2 Схема базы данных в MySQL

3.3. Пример работы программы

Рассмотрим процесс работы программы на простом примере. Допустим, необходимо провести по 15 одинаковых занятий (назовём их от «*pair 1*» до «*pair 15*») в каждой из трёх групп. Каждый предмет для всех трёх групп ведёт один из пяти преподавателей в одном из семи кабинетов. Для хранения информации используем таблицу базы данных (рис. 3.3), в которой

- *id* – идентификатор занятия
- *teacher_id* – идентификатор преподавателя
- *group_num* – номер группы, в которой необходимо провести занятие
- *room* – номер кабинета, в котором проходит занятие
- *class_name* – название занятия

Далее, запрашивая по очереди все занятия с каждым из преподавателей и с каждым из кабинетов, составляем граф с 45 вершинами и 504 рёбрами. Сформировав граф, запускаем на нём алгоритм раскраски. Алгоритм выдал оптимальный вариант раскраски – 15 цветов (меньше невозможно, т. к. количество занятий в каждой группе 15).

id	teacher_id	group_num	room	class_name
1	3	1	7	pair 1
2	3	2	7	pair 1
3	3	3	7	pair 1
4	1	1	7	pair 2
5	1	2	7	pair 2
6	1	3	7	pair 2
7	5	1	4	pair 3
8	5	2	4	pair 3
9	5	3	4	pair 3
10	4	1	6	pair 4
11	4	2	6	pair 4
12	4	3	6	pair 4
13	5	1	6	pair 5
14	5	2	6	pair 5
15	5	3	6	pair 5
16	1	1	1	pair 6
17	1	2	1	pair 6
18	1	3	1	pair 6
19	3	1	7	pair 7
20	3	2	7	pair 7
21	3	3	7	pair 7
22	2	1	1	pair 8
23	2	2	1	pair 8
24	2	3	1	pair 8
25	3	1	7	pair 9
26	3	2	7	pair 9
27	3	3	7	pair 9
28	2	1	5	pair 10
29	2	2	5	pair 10
30	2	3	5	pair 10
31	5	1	3	pair 11
32	5	2	3	pair 11
33	5	3	3	pair 11
34	4	1	4	pair 12
35	4	2	4	pair 12
36	4	3	4	pair 12
37	3	1	6	pair 13
38	3	2	6	pair 13
39	3	3	6	pair 13
40	2	1	6	pair 14
41	2	2	6	pair 14
42	2	3	6	pair 14
43	1	1	7	pair 15
44	1	2	7	pair 15
45	1	3	7	pair 15

Рис. 3.3 Содержимое таблицы занятий в БД

Предположим, что в день проводится максимум 4 занятия:

- 9:30 – 11:05 – первая пара
- 11:15 – 12:50 – вторая пара
- 13:40 – 15:15 – третья пара
- 15:25 – 17:00 – четвёртая пара

Тогда все запланированные занятия можно провести за 4 дня, например, с понедельника по четверг. После подстановки занятий вместо соответствующих вершин получаем расписание, приведённое в таблице ниже.

	Группа 1	Группа 2	Группа 3
ПН 9:30 - 11:05	pair 13	pair 11	pair 8
ПН 11:15 - 12:50	pair 1	pair 6	pair 10
ПН 13:40 - 15:15	pair 10	pair 1	pair 5
ПН 15:25 - 17:00	pair 8	pair 15	pair 1
ВТ 9:30 - 11:05	pair 7	pair 5	pair 14
ВТ 11:15 - 12:50	pair 5	pair 7	pair 4
ВТ 13:40 - 15:15	pair 3	pair 4	pair 7
ВТ 15:25 - 17:00	pair 9	pair 14	pair 3
СР 9:30 - 11:05	pair 15	pair 9	pair 11

СР 11:15 - 12:50	pair 11	pair 2	pair 9
СР 13:40 - 15:15	pair 6	pair 13	pair 12
СР 15:25 - 17:00	pair 2	pair 12	pair 13
ЧТ 9:30 - 11:05	pair 14	pair 3	pair 2
ЧТ 11:15 - 12:50	pair 4	pair 10	pair 6
ЧТ 13:40 - 15:15	pair 12	pair 8	pair 15

Заключение

Составление расписаний является неотъемлемой частью повседневной жизни. Зачастую при составлении расписания изначально известно только то, что необходимо сделать и что нельзя сделать одновременно. Оптимально составленное расписание способно принести большую выгоду, но при этом поиск такого расписания может занять длительное время. Для ускорения процесса необходим способ решать данную задачу с использованием современных технологий.

Одним из способов нахождения расписания является сведение этой задачи к важной задаче теории графов – задаче о раскраске графа. Поиск методов эффективного решения последней продолжается и сейчас.

В данной работе проведен анализ алгоритма вершинной раскраски графа, предложенный Красновой А. Ю., предложен новый алгоритм вершинной раскраски графа и проведено сравнение работы рассмотренных алгоритмов раскраски на коллекции графов. Полученный алгоритм раскраски вершин графа был применен к решению задачи составления расписаний и осуществлена программная реализация для демонстрации работоспособности предложенного подхода.

Список литературы

1. Краснова Александра Юрьевна. Матрицы инциденций и раскраски графа: диссертация ... кандидата физико-математических наук: 01.01.09 / Краснова Александра Юрьевна; [Место защиты: С.-Петерб. гос. ун-т]. - Санкт-Петербург, 2009, 87 с.: ил.
2. Глоссарий теории графов на Википедии. https://ru.wikipedia.org/wiki/Глоссарий_теории_графов
3. Оре О. Графы и их применение. М.: Мир, 1965, 174 стр.
4. Татт У. Теория графов. М.: Мир, 1988, 424 стр.
5. Харари Ф. Теория графов. М.: Мир, 1973, 300 стр.
6. Раскраска графов на Википедии. https://ru.wikipedia.org/wiki/Раскраска_графов
7. Бацын М. В., Комоско Л. Ф. Быстрый алгоритм для решения задачи о раскраске графа с использованием битовых операций // Труды 38-й конференции «Информационные технологии и системы – 2014» Н. Новгород: ИППИ РАН, 2014. С. 432-438.
8. Coloring Problems. DIMACS Graph Format. http://prolland.free.fr/works/research/dsat/dimacs.html
9. Graph Coloring Instances. http://mat.gsia.cmu.edu/COLOR/instances.html

Приложение

Приложение 1

```
#include<iostream>
#include<fstream>
#include<vector>
#include<string>

using namespace std;

int** multiply(int size, int ** A, int ** B)
{
	int** X = new int*[size];
	for (int i = 0; i < size; ++i)
		X[i] = new int[size];
	for (int i = 0; i < size; ++i)
		for (int j = 0; j < size; ++j)
			X[i][j] = 0;

	for (int i = 0; i < size; ++i)
		for (int j = 0; j < size; ++j)
			for (int k = 0; k < size; ++k)
				X[i][j] += A[i][k] * B[k][j];

	return X;
}
int** transp(int size, int** A)
{
	int** X = new int*[size];
	for (int i = 0; i < size; ++i)
		X[i] = new int[size];
	for (int i = 0; i < size; ++i)
		for (int j = 0; j < size; ++j)
			X[i][j] = A[j][i];
	return X;
}

char** mix(int vertexNumber, char** oldGraph, vector<int> groups, int groupNumber)
{
	vector<vector<int>> groupMembers(groupNumber);

	for (int i = 0; i < vertexNumber; ++i)
		groupMembers[groups[i]].push_back(i);

	vector<int> newOrder;
	newOrder.clear();
	for (int i = 0; i < groupMembers.size(); ++i)
		for (int j = 0; j < groupMembers[i].size(); ++j)
			newOrder.push_back(groupMembers[i][j]);

	int** p = new int*[vertexNumber];
	for (int i = 0; i < vertexNumber; ++i)
		p[i] = new int[vertexNumber];

	for (int i = 0; i < vertexNumber; ++i)
		for (int j = 0; j < vertexNumber; ++j)
			p[i][j] = 0;

	for (int i = 0; i < vertexNumber; ++i)
		p[i][vertexNumber - newOrder[i] - 1] = 1;

	int** newGraph = new int*[vertexNumber];
	for (int i = 0; i < vertexNumber; ++i)
		newGraph[i] = new int[vertexNumber];
```

```
	for (int i = 0; i < vertexNumber; ++i)
		for (int j = 0; j < vertexNumber; ++j)
			newGraph[i][j] = oldGraph[i][j] - '0';

	int** x = multiply(vertexNumber, p, newGraph);
	int** pp = transp(vertexNumber, p);

	int** newNewGraph = multiply(vertexNumber, x, pp);

	for (int i = 0; i < vertexNumber; ++i)
		for (int j = 0; j < vertexNumber; ++j)
			oldGraph[i][j] = '0' + newNewGraph[i][j];

	for (int i = 0; i < vertexNumber; ++i)
		delete[] x[i];
	delete[] x;
	for (int i = 0; i < vertexNumber; ++i)
		delete[] pp[i];
	delete[] pp;
	for (int i = 0; i < vertexNumber; ++i)
		delete[] newGraph[i];
	delete[] newGraph;
	for (int i = 0; i < vertexNumber; ++i)
		delete[] newNewGraph[i];
	delete[] newNewGraph;
	for (int i = 0; i < vertexNumber; ++i)
		delete[] p[i];
	delete[] p;

	return oldGraph;
}

int main()
{
	setlocale(LC_ALL, "rus");
	char filename[256];
	//Graph information loading
	int vertexNumber = 0;

	ifstream fin;
	cout << "Введите имя файла" << endl;
	cin >> filename;
	fin.open(filename);

	if (!fin.is_open())
	{
		cout << "Не удалось открыть файл!" << endl;
		system("pause");
		return 1;
	}

	int edgeNumber = 0, ii = 0, jj = 0;

	char** graph = NULL;
	char comment[256];

	while (fin.good())
	{
		char c = fin.get();
		switch (c)
		{
		case 'c':
			fin.getline(comment, 256);
			break;
		case 'p':
			fin.get();
			fin.ignore(10, ' ');
			fin >> vertexNumber;
```

```
                        fin >> edgeNumber;

                        graph = new char*[vertexNumber];
                        for (int i = 0; i < vertexNumber; ++i)
                                graph[i] = new char[vertexNumber];
                        for (int i = 0; i < vertexNumber; ++i)
                                for (int j = 0; j < vertexNumber; ++j)
                                        graph[i][j] = '0';
                        break;
                case 'e':
                        if (graph == NULL)
                        {
                                cout << "Invalid DIMACS file!" << endl;
                                system("pause");
                                return 2;
                        }
                        fin >> ii; fin >> jj;
                        if (vertexNumber > 0)
                        {
                                graph[ii - 1][jj - 1] = '1';
                                graph[jj - 1][ii - 1] = '1';
                        }
                        break;
                }
        }

        fin.close();

        int rounds = 0;
        cout << "Введите кол-во повторений" << endl;
        cin >> rounds;

        char** currGraph = new char*[vertexNumber];
        for (int i = 0; i < vertexNumber; ++i)
                currGraph[i] = new char[vertexNumber];
        for (int i = 0; i < vertexNumber; ++i)
                for (int j = 0; j < vertexNumber; ++j)
                        currGraph[i][j] = graph[i][j];

        //additional stuff
        int bestGroupCount = INT_MAX;
        vector<int> bestGroups;

        vector<int> groups(vertexNumber, -1);

        int currGroup = -1;
        int finishedVertexes = 0;
        vector<int> currGroupMembers;
        bool is_fit = true;

        //here we go!
        for (int round = 0; round < rounds; ++round)
        {
                //reset
                groups.assign(vertexNumber, -1); currGroup = -1; finishedVertexes = 0; is_fit =
true;

                while (finishedVertexes < vertexNumber)
                {
                        ++currGroup;
                        currGroupMembers.clear();
                        is_fit = true;
                        for (int i = 0; i < vertexNumber; ++i, is_fit = true)
                        {
                                if (groups[i] != -1) //already assigned
                                        continue;
                                for (int j = 0; j < currGroupMembers.size(); ++j)
                                        if (currGraph[i][currGroupMembers[j]] == '1') //connected
to another vertex in group
```

```
                                        is_fit = false;
                                if (is_fit)
                                {
                                        groups[i] = currGroup;
                                        ++finishedVertexes;
                                        currGroupMembers.push_back(i);
                                }
                        }
                }

                //improvement check
                if (currGroup + 1 < bestGroupCount)
                {
                        bestGroupCount = currGroup + 1;
                        bestGroups = groups;
                }

                //mixing vertexes
                currGraph = mix(vertexNumber, currGraph, groups, currGroup + 1);
        }

        vector<vector<int>> groupMembers(currGroup + 1);

        for (int i = 0; i < vertexNumber; ++i)
                groupMembers[bestGroups[i]].push_back(i);

        //output
        for (int i = 0; i < bestGroupCount; ++i)
        {
                cout << "Group " << i + 1 << ": ";
                for (int j = 0; j < groupMembers[i].size(); ++j)
                        cout << groupMembers[i][j] + 1 << ' ';
                cout << endl;
        }

        //cleaning
        for (int i = 0; i < vertexNumber; ++i)
                delete[] graph[i];
        delete[] graph;

        for (int i = 0; i < vertexNumber; ++i)
                delete[] currGraph[i];
        delete[] currGraph;

        system("pause");
        return 0;
}
```

Приложение 2

```
#include<iostream>
#include<fstream>
#include<vector>
#include<utility>

using namespace std;

class Vertex
{
public:
        int number;
        int degree;
        vector<int> lockedColors;

        Vertex()
```

```
	{
		number = 0;
		degree = 0;
		lockedColors = vector<int>();
	};
	Vertex(char** graph, int vertexNumber, int index)
	{
		number = index;
		degree = 0;
		for (int i = 0; i < vertexNumber; ++i)
			if (graph[index][i] == '1')
				++degree;
		lockedColors = vector<int>();
	}
};
bool operator < (Vertex A, Vertex B)
{
	if (A.lockedColors.size() < B.lockedColors.size())
		return true;
	else if (A.lockedColors.size() > B.lockedColors.size())
		return false;
	else if (A.degree < B.degree)
		return true;
	else return false;
}
bool operator <= (Vertex A, Vertex B)
{
	if (A.lockedColors.size() < B.lockedColors.size())
		return true;
	else if (A.lockedColors.size() > B.lockedColors.size())
		return false;
	else if (A.degree <= B.degree)
		return true;
	else return false;
}
bool operator > (Vertex A, Vertex B)
{
	if (A.lockedColors.size() > B.lockedColors.size())
		return true;
	else if (A.lockedColors.size() < B.lockedColors.size())
		return false;
	else if (A.degree > B.degree)
		return true;
	else return false;
}
bool operator >= (Vertex A, Vertex B)
{
	if (A.lockedColors.size() > B.lockedColors.size())
		return true;
	else if (A.lockedColors.size() < B.lockedColors.size())
		return false;
	else if (A.degree >= B.degree)
		return true;
	else return false;
}
ostream & operator << (ostream &os, const Vertex A)
{
	os << A.number;
	return os;
}

void quickSort(int l, int r, vector<Vertex> &a, vector<int> &indexes)
{
	Vertex x = a[(l + r) / 2];

	int i = l;
	int j = r;
	while (i <= j)
	{
```

```
                while (a[i] < x) i++;
                while (a[j] > x) j--;
                if (i <= j)
                {
                        swap(a[i], a[j]);
                        swap(indexes[a[i].number], indexes[a[j].number]);
                        i++;
                        j--;
                }
        }
        if (i < r)
                quickSort(i, r, a, indexes);

        if (l < j)
                quickSort(l, j, a, indexes);
}

void addColor(vector<vector<int>> &colors, vector<int> &CF)
{
        colors.push_back(vector<int>());
        colors.back().push_back(0);
        CF.push_back(CF.size());
}

int main()
{
        setlocale(LC_ALL, "rus");
        char filename[256];
        //Graph information loading
        int vertexNumber = 0;

        ifstream fin;
        cout << "Введите имя файла" << endl;
        cin >> filename;
        fin.open(filename);

        if (!fin.is_open())
        {
                cout << "Не удалось открыть файл!" << endl;
                system("pause");
                return 1;
        }

        int edgeNumber = 0, ii = 0, jj = 0;

        char** graph = NULL;
        char comment[256];

        while (fin.good())
        {
                char c = fin.get();
                switch (c)
                {
                case 'c':
                        fin.getline(comment, 256);
                        break;
                case 'p':
                        fin.get();
                        fin.ignore(10, ' ');
                        fin >> vertexNumber;
                        fin >> edgeNumber;

                        graph = new char*[vertexNumber];
                        for (int i = 0; i < vertexNumber; ++i)
                                graph[i] = new char[vertexNumber];
                        for (int i = 0; i < vertexNumber; ++i)
                                for (int j = 0; j < vertexNumber; ++j)
                                        graph[i][j] = 0;
                        break;
```

```
		case 'e':
			fin >> ii; fin >> jj;
			if (vertexNumber > 0)
			{
				graph[ii - 1][jj - 1] = '1';
				graph[jj - 1][ii - 1] = '1';
			}
			break;
		}
	}

	fin.close();

	vector<Vertex> vertexes;
	vertexes.reserve(vertexNumber);
	for (int i = 0; i < vertexNumber; ++i)
		vertexes.push_back(Vertex(graph, vertexNumber, i));

	//additional data
	vector<int> newIndexes;
	newIndexes.reserve(vertexNumber);
	for (int i = 0; i < vertexNumber; ++i)
		newIndexes.push_back(i);

	vector<vector<int>> colors;
	vector<int> colorFrequency;
	addColor(colors, colorFrequency);

	//initial quick vertex sorting
	quickSort(0, vertexNumber - 1, vertexes, newIndexes);


	bool isLocked = false;
	int currentColor = 0;
	bool toAdd = true;
	//Coloring
	while (!vertexes.empty())
	{
		for (int i = colorFrequency.size() - 1; i >= 0; --i)
		{
			currentColor = colorFrequency[i];
			//color searching
			isLocked = false;
			for (int j = 0; j < vertexes.back().lockedColors.size(); ++j)
			{
				if (currentColor == vertexes.back().lockedColors[j])
					isLocked = true;
			}
			//if color found
			if (!isLocked)
			{
				//adding vertex to list
				colors[currentColor].push_back(vertexes.back().number);
				++colors[currentColor][0];
				//connected vertexes correction
				for (int n = 0; n < vertexNumber; ++n)
				{
					if (graph[vertexes.back().number][n] == '1')
					{
						//lockedColor addition nesessity checking (no
repeat)
						if (vertexes[newIndexes[n]].lockedColors.size() ==
0)
							toAdd = true;
						else
						{
							toAdd = true;
							for (int j = 0; j <
vertexes[newIndexes[n]].lockedColors.size(); ++j)
```

```
                                                {
                                                        if (currentColor ==
vertexes[newIndexes[n]].lockedColors[j])
                                                        {
                                                                toAdd = false;
                                                                break;
                                                        }
                                                }
                                        }
                                        //Addition of an adjacent color
                                        if (toAdd)
                                        {

        vertexes[newIndexes[n]].lockedColors.push_back(currentColor);
                                                //vertexes resorting
                                                while (newIndexes[n] + 2 < vertexes.size()
&& vertexes[newIndexes[n]] > vertexes[newIndexes[n] + 1])
                                                {
                                                        swap(vertexes[newIndexes[n]],
vertexes[newIndexes[n] + 1]);

        swap(newIndexes[vertexes[newIndexes[n]].number], newIndexes[vertexes[newIndexes[n] +
1].number]);
                                                }
                                        }
                                }
                        }

                        //colors resorting
                        int k = 0;
                        while (k < i)
                        {
                                if (colors[colorFrequency[i - k]][0] >
colors[colorFrequency[i - k - 1]][0])
                                {
                                        swap(colorFrequency[i - k], colorFrequency[i - k -
1]);
                                        ++k;
                                }
                                else break;
                        }

                        //Vertex deleting
                        for (int m = 0; m < vertexNumber; ++m)
                        {
                                graph[m][vertexes.back().number] = '0';
                                graph[vertexes.back().number][m] = '0';
                        }
                        vertexes.pop_back();
                        break;
                }
                //if no available colors
                if (isLocked && i == 0)
                {
                        addColor(colors, colorFrequency);
                        break;
                }
        }
    }

    //output: "groupNumber: V1, V2..."
    for (int i = 0; i < colors.size(); ++i)
    {
        cout << "Group " << i + 1 << ":";
        for (int j = 1; j < colors[i].size(); ++j)
                cout << ' ' << colors[i][j] + 1;
        cout << endl;
    }
```

```
	//cleaning
	for (int i = 0; i < vertexNumber; ++i)
		delete[] graph[i];
	delete[] graph;

	system("pause");
	return 0;
}
```