

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ
ФЕДЕРАЦИИ

ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО
ОБРАЗОВАНИЯ “МОСКОВСКИЙ ФИЗИКО-ТЕХНИЧЕСКИЙ ИНСТИТУТ
(ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ)”

ФАКУЛЬТЕТ ИННОВАЦИЙ И ВЫСОКИХ ТЕХНОЛОГИЙ
КАФЕДРА ДИСКРЕТНОЙ МАТЕМАТИКИ

Выпускная квалификационная работа по направлению
01.03.02 "Прикладные математика и информатика"
НА ТЕМУ:

**ВЕРИФИКАЦИЯ ДОКАЗАТЕЛЬСТВА ТЕОРЕМЫ О НИЖНЕЙ
ОЦЕНКЕ ХРОМАТИЧЕСКОГО ЧИСЛА ПЛОСКОСТИ
В СИСТЕМЕ Coq**

Студент _____ Анюшева Е.Б.

Научный руководитель к.ф.-м.н. _____ Дашков Е.В.

МОСКВА, 2019

Оглавление

	Стр.
Аннотация	3
Введение	4
0.1 Хроматическое число плоскости. Задача Нелсона — Эрдёша — Хадвигера	4
0.2 Система Coq . Описание, история, возможности, применения . . .	5
Глава 1. Реализация графа в Coq	7
1.1 Представление графов в Coq	7
1.2 Доказательства корректности операций над графами	12
Глава 2. Доказательство свойств раскраски малых графов в Coq	14
2.1 Язык <code>Ltac</code> , <code>pattern matching</code> и <code>goal matching</code>	15
2.2 Типы возможных правильных раскрасок графа T в не более чем 4 цвета	15
2.3 Типы возможных правильных раскрасок графа H в не более чем 4 цвета	18
Глава 3. Заключение	21
3.1 Выводы	21
3.2 Планы будущей работы	21
Глава 4. Приложение	22

Аннотация

В ходе работы разработаны методы работы с графами в системе **Coq**, а также методы работы с правильными раскрасками графов. Реализованы некоторые графы, представленные в статье А. de Grey, «The chromatic number of the plane is at least 5» [1], а также формализованы и доказаны в системе **Coq** утверждения о типах возможных правильных раскрасок графов Т и Н. Полный код работы расположен по адресу https://github.com/LenaAn/deGrey_proofs.

Введение

0.1 Хроматическое число плоскости. Задача Нелсона — Эрдёша — Хадвигера

Граф G — это упорядоченная пара $G := (V, E)$, где V — непустое множество, а E — подмножество $V \times V$. Если $(u, v) \in E$, то вершины u и v называются *смежными*. Обозначение $u \sim v$.

Раскраска f графа G — это отображение из V в множество цветов. Раскраска f называется *правильной*, если $u \sim v \rightarrow f(u) \neq f(v)$.

Хроматическое число графа — это минимальное количество цветов, в которые можно правильно раскрасить граф.

Граф единичных расстояний — это граф, вершинами которого являются некоторые точки евклидовой плоскости, а ребрами соединены все пары вершин, находящиеся на расстоянии 1.

Хроматическое число плоскости χ — это минимальное число цветов χ , в которое можно правильно раскрасить любой граф единичных расстояний.

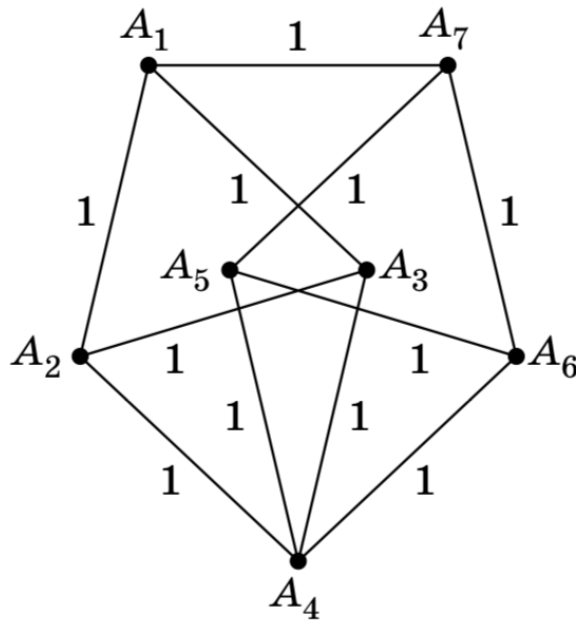


Рисунок 1 — Веретено Мозера

Задача Нелсона — Эрдёша — Хадвигера заключается в нахождении хроматического числа плоскости. С 1950 года известно [4], что хроматическое число плоскости хотя бы 4 и не больше 7.



Рисунок 2 — Раскраска плоскости в 7 цветов

На рисунке 1 приведен пример графа, который не красится в 3 цвета, а на рисунке 2 приведено замощение плоскости в 7 цветов так, что никакая пара точек на расстоянии 1 не окрашена в один цвет.

В апреле 2018 года Обри де Грей опубликовал статью [1], в которой доказал, что хроматическое число плоскости хотя бы 5. На момент написания работы задача является открытой. Данная работа фокусируется на уточнении неясных мест в данной статье, явных детерминированных конструкциях графов из статьи и верификации отдельных утверждений статьи в системе Coq.

0.2 Система Coq. Описание, история, возможности, применения

Coq — это формальная система управления доказательствами. Она предоставляет формальный язык *Gallina* для написания математических определений, исполняемых алгоритмов и теорем со средой для полуинтерактивной разработки верифицированных доказательств. Пользователь интерактивно создаёт доказательство сверху вниз, начиная с цели (то есть от гипотезы, которую необходимо доказать). Coq содержит *тактики*, которые позволяют автомати-

зировать доказательства теорем. Язык **Ltac** позволяет пользователю самому определять новые тактики.

Примеры использования системы:

- *CompCert*: оптимизирующий компилятор для почти всего языка программирования *C*, который полностью форматизован и проверен на **Coq**.
- Структура данных с несвязным множеством: доказательство корректности в **Coq** было опубликовано в 2007 году.
- Теорема Фейта – Томпсона: формальное доказательство с использованием **Coq** было завершено в сентябре 2012 года.
- Теорема о четырех цветах: формальное доказательство с использованием **Coq** было завершено в 2005 году.

Глава 1. Реализация графа в Coq

1.1 Представление графов в Coq

Реализации графов из статьи де Грея приведены в файле `myGraphs.v` 4.1. Представление графа было взято из книги Softwarefoundations [2]. Для представления графа используются модули `FSets` и `FMaps`, которые предоставляют интерфейсы множества и отображения. Эти модули принимают различные типы ключей, в данном случае мы будем использовать тип позитивных чисел `positive` из модуля `PositiveOrderedTypeBits`.

```
Module E := PositiveOrderedTypeBits.
Module S <: FSetInterface.S := PositiveSet.
Module M <: FMapInterface.S := PositiveMap.
```

Вершина `node` – это элемент типа `positive`, `nodemap` – это отображение из вершин, а граф `graph` – это отображение из типа вершина в тип множество вершин. Тип `positive` был выбран из-за того, что в нем оператор сравнения определен так, чтобы поиск по ключу типа `positive` в множестве и отображении был более эффективным.

```
Definition node := E.t.
Definition nodeset := S.t.
Definition nodemap: Type -> Type := M.t.
Definition graph := nodemap nodeset.
```

Для работы с графами были определены функции добавления ребра в существующий граф и построения графа из списка ребер.

```
Definition add_edge (e: (E.t*E.t)) (g: graph) : graph :=
  M.add (fst e) (S.add (snd e) (adj g (fst e)))
  (M.add (snd e) (S.add (fst e) (adj g (snd e)))) g).
```

В данной функции ребро представляется парой вершин. В данной работе реализуются неориентированные графы без петель.

```
Definition mk_graph (el: list (E.t*E.t)) :=
  fold_right add_edge (M.empty _) el.
```

В терминах определенных выше функций построение графа K3 выглядит следующим образом:

```
Definition K3 :=
  mk_graph [ (1, 2) ; (2, 3); (1, 3)].
```

Далее для работы с графом можно использовать функцию вывода множества вершин и функцию `gr_show` вывода множества ребер.

```
Compute (S.elements (Mdomain K3)).
```

```
(*
  = [2; 1; 3]
    : list S.elc.
*)
```

```
Function gr_show (g : graph) : list (node * node) :=
  S.fold
    (fun n l => (map (fun y => (n, y)) (S.elements (adj g n))) ++ l)
    (Mdomain g) nil.
```

```
Compute gr_show K3.
```

```
(*
= [(3, 2); (3, 1); (1, 2); (1, 3); (2, 1); (2, 3)]
  : list (node * node)
*)
```

Для построения графов разработаны несколько вспомогательных функций.

Рекурсивная функция `l_rng` находит минимум и максимум в списке, функция `gr_rng` находит в графе минимальный и максимальный номера вершин. Функция `rename_all` принимает на вход граф `G` и функцию `f` из номеров вершин в номера вершин и выдает граф, полученный применением `f` к вершинам графа `G`. Функция `delete_edge` удаляет ребро из графа, а функция `rename_in_order` переименовывает вершины графа в отрезок от 1 до количества вершин, сохраняя при этом из порядок.

Основные функции, используемые для построения графов – это `mk_art`, которая соединяет два графа по вершине, `mk_cmn_edge`, которая соединяет два графа по ребру и `add_edges`, которая добавляет ребра в граф.

В терминах описанных функций конструкция графа Н на основе графа КЗ выглядит следующим образом:

```
Definition Н : graph :=
  let g1 := rename_in_order (mk_cmn_edge КЗ КЗ 1 3 1 3) in
  let g2 := rename_in_order
    (mk_cmn_edge g1 КЗ 1 (snd (gr_rng g1)) 1 3) in
  let g3 := rename_in_order
    (mk_cmn_edge g2 КЗ 1 (snd (gr_rng g2)) 1 3) in
  let g4 := rename_in_order
    (mk_cmn_edge g3 КЗ 1 (snd (gr_rng g3)) 1 3) in
  rename_in_order (add_edge (2, snd (gr_rng g4)) g4).
```

Т. е. граф Н построен из 5 копий КЗ, склеенных друг с другом по ребру, и добавленного ребра.

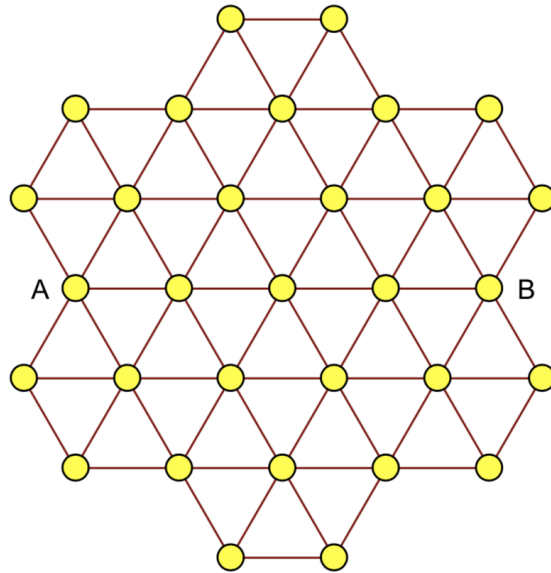


Рисунок 1.1 — Граф J

Граф J состоит из 6 копий Н, склеенных между собой по ребру.

```
Definition J: graph :=
  let HH := mk_cmn_edge Н Н 2 3 6 7 in
```

```

let HH_H := mk_cmn_edge HH H 7 2 6 7 in
let HHH := rename_node 14 12 HH_H in
let HHH_H := mk_cmn_edge HHH H 6 7 6 7 in
let HHHH := rename_node 19 17 HHH_H in
let HHHH_H := mk_cmn_edge HHHH H 5 6 6 7 in
let HHHHH := rename_node 24 22 HHHH_H in
let HHHHH_H := mk_cmn_edge HHHHH H 4 5 6 7 in
let HHHHHH := rename_node 29 27 HHHHH_H in
let HHHHHH_H := mk_cmn_edge HHHHHH H 3 4 6 7 in
let HHHHHHH := rename_node 34 32 HHHHHH_H in
rename_in_order (rename_node 37 9 HHHHHHH).

```

В графе J вершины, находящиеся на расстоянии 2 от центра (в данной реализации вершины 9, 12, 16, 20, 24, 28) называются *соединяющими вершинами*.

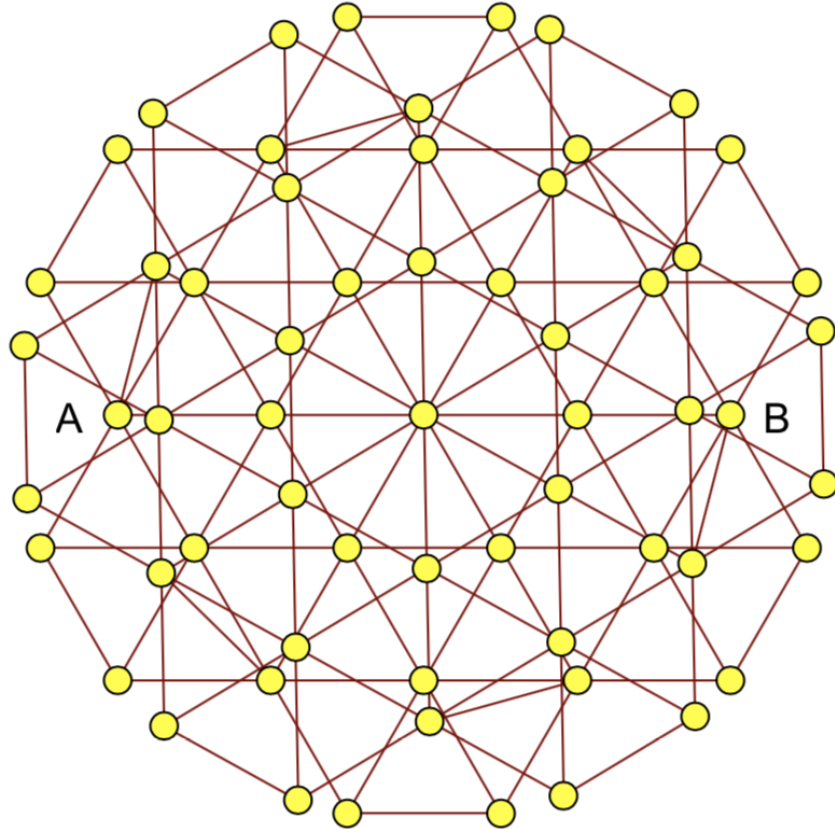


Рисунок 1.2 — Граф К

Граф К состоит из двух копий графа J, соединенных по центру, и ребер между соответствующими *соединяющими вершинами*.

Definition K: graph :=

```
let JJ := mk_art J J 1 1 in
```

```
let JJ := add_edges [(9, 9+31); (12, 12+31); (16, 16+31); (20, 20+31);  
                    (24, 24+31); (28, 28+31)] JJ in
```

```
rename_in_order JJ.
```

Наконец, граф L состоит из двух копий графа K, склеенных между собой по *соединяющей вершине* и ребра между соединяющими вершинами, являющимися противоположными точке соединения.

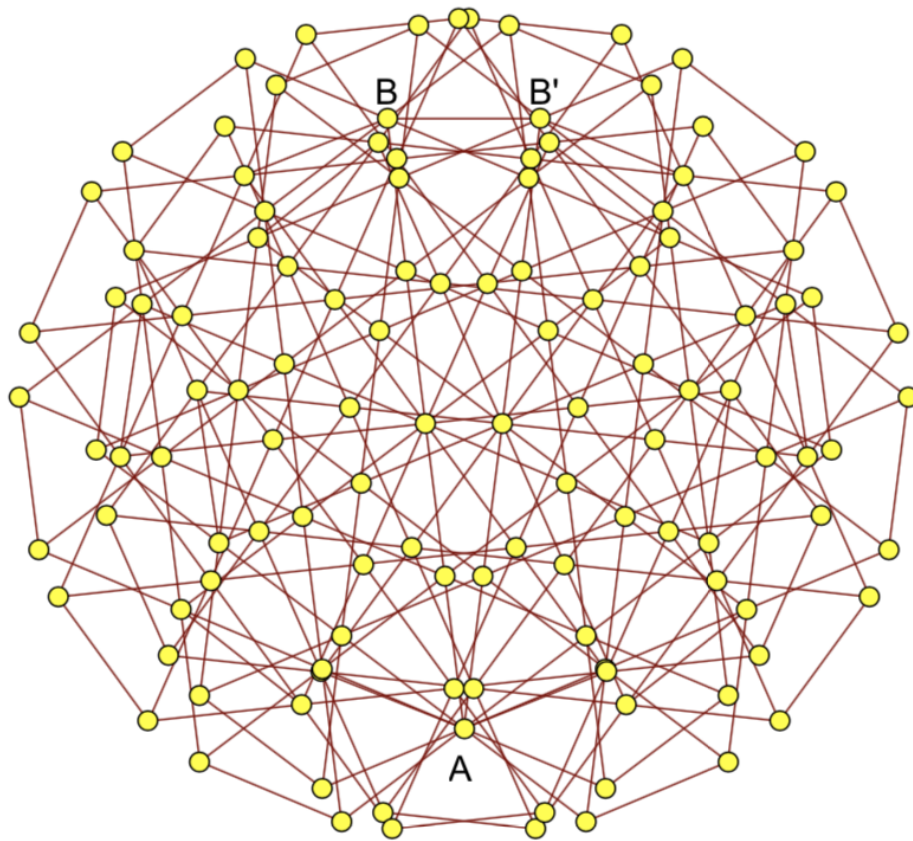


Рисунок 1.3 — Граф L

1.2 Доказательства корректности операций над графами

В файле `myGraphs_Properties.v` 4.2 представлены доказательства корректности определенных ранее графов, т. е. что они являются неориентированными графами без петель.

```
Definition graph_ok (g : graph) :=
  undirected g /\ no_selfloop g.
```

Для доказательства корректности графов введены и доказаны вспомогательные леммы `adj_M_In` и `edge_corr_1`.

```
Lemma adj_M_In : forall g n m,
  S.In m (adj g n) -> M.In n g.
```

```
Lemma edge_corr_1 : forall g n m, edge g n m -> S.In m (nodes g).
```

А также определена тактика `gr_destr`.

```
Ltac gr_destr h :=   apply S.elements_1 in h; compute in h;
  repeat rewrite InA_cons in h; rewrite InA_nil in h;
  repeat destruct h as [? | h]; try inversion h; subst.
```

Данная тактика из посылки, что `i` – вершина графа, заключает, что `i` равняется какому-то числу от 1 до количества вершин и использует инверсию на этой посылке, тем самым перебирая все возможные вершины графа.

С использованием этой тактики доказательство корректности выглядит одинаково для любого построенного выше графа, приведем его для `H`:

```
Lemma H_ok : graph_ok H.
Proof.
split.
- unfold undirected. intros. remember H as H'.
  clear HeqH'. apply edge_corr_1 in H.
  gr_destr H; gr_destr H'; reflexivity.
- unfold no_selfloop. repeat intro. remember H as H'.
```

```
clear HeqH'. apply edge_corr_1 in H. gr_destr H; gr_destr H';  
discriminate.
```

Qed.

Глава 2. Доказательство свойств раскраски малых графов в Soq

Назовем две раскраски f, f' графа G *существенно одинаковыми*, если существуют изоморфизм графа g и перестановка цветов ω такие, что $f(v) = \omega(f'(g(v)))$. Также назовем две раскраски *существенно различными*, если они не являются существенно одинаковыми.

Граф H – это граф

$$H := (\{1, 2, 3, 4, 5, 6, 7\},$$

$$\{(1, 2), (1, 3), (1, 4), (1, 5), (1, 6), (1, 7),$$

$$(2, 3), (3, 4), (4, 5), (5, 6), (6, 7), (7, 2)\})$$

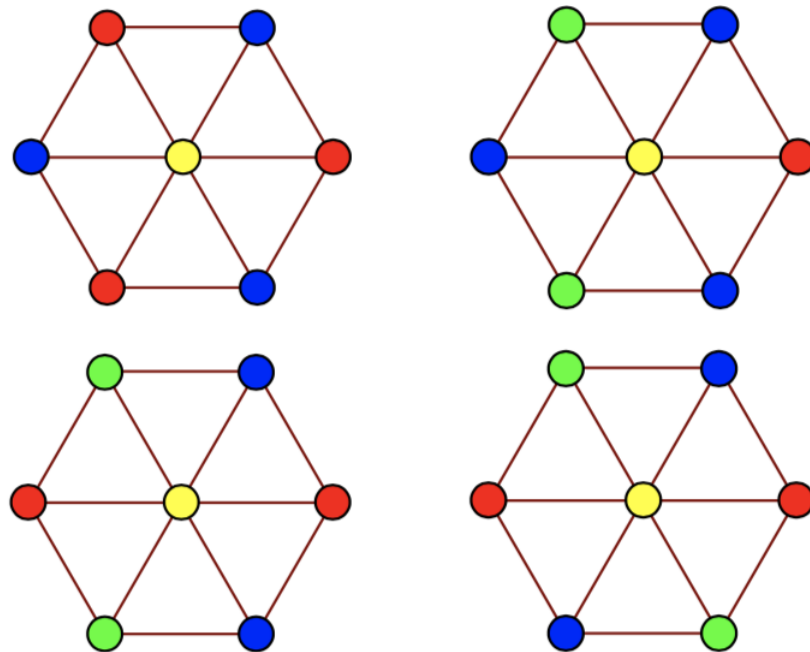


Рисунок 2.1 — Существенно различные способы раскрасить граф H в не более чем 4 цвета

В статье [1] утверждается что существует только 4 существенно различные раскраски графа H в не более чем в 4 цвета. Это утверждение обосновывается перебором вариантов наличия или отсутствия монохроматических троек.

Описание типов для работы с раскрасками находятся в файле `my_New_Coloring.v` 4.3. В нем описан индуктивный тип `is_color`, который имеет 4 конструктора – по номерам возможных цветов в раскраске. Тот факт, что этот тип индуктивный, позволяет использовать инверсию по этому типу.

2.1 Язык Ltac, pattern matching и goal matching

В данной главе активно используется язык Ltac и инструмент goal matching, представляемый этим языком. Язык Ltac, впервые представленный в статье «A Tactic Language for the System Coq» [7], предоставляет оператор соответствия (*pattern matching*) не только для термов, но и для контекста доказательства (*goal matching*), т. е. цели и посылок. Данный оператор позволяет автоматизировать применение низкоуровневых тактик и значительно сократить длину доказательства.

2.2 Типы возможных правильных раскрасок графа T в не более чем 4 цвета

Назовем *тройкой* любой граф, изоморфный графу T на четырех вершинах

$$T := (\{1, 2, 3, 4\}, \{(1, 2), (1, 3), (1, 4)\})$$

.

Утверждение: Существует только три существенно различные раскраски графа G , если граф G изоморфен T .

В системе Coq это раскраски можно описать следующим образом:

```
(* Monochromatic *)
Definition type1_triple (el: list node) (c: Coloring) :=
  let center := nth 0 el 1 in
  let v1 := nth 1 el 1 in
```

```

let v2 := nth 2 el 1 in
let v3 := nth 3 el 1 in
let c1 := c center in
let c2 := c v1 in
~ (c1 = c2) /\ same_color c v1 v2 /\ same_color c v2 v3.

```

(* 2 and 1 *)

```

Definition type2_triple (el: list node) (c: Coloring) :=
  let center := nth 0 el 1 in
  let v1 := nth 1 el 1 in
  let v2 := nth 2 el 1 in
  let v3 := nth 3 el 1 in

  let c1 := c center in
  let c2 := c v1 in
  let c3 := c v2 in
  let c4 := c v3 in
  ~ (c1 = c2) /\ ~ (c1 = c3) /\ ~ (c1 = c4) /\
    ( (c2 = c3 /\ ~ c2 = c4) \/ (c2 = c4 /\ ~ c2 = c3) \/
      (c3 = c4 /\ ~ c3 = c2) ).

```

(* All 3 different *)

```

Definition type3_triple (el: list node) (c: Coloring) :=
  let center := nth 0 el 1 in
  let v1 := nth 1 el 1 in
  let v2 := nth 2 el 1 in
  let v3 := nth 3 el 1 in

  let c1 := c center in
  let c2 := c v1 in
  let c3 := c v2 in
  let c4 := c v3 in
  ~ (c1 = c2) /\ ~ (c1 = c3) /\ ~ (c1 = c4) /\
    (~ c2 = c3) /\ (~ c2 = c4) /\ (~ c3 = c4).

```


Каждая функция имеет тип `Prop`, принимает на вход список вершин и раскраску, при этом первый элемент в списке – номер вершины, соединенной со всеми остальными. Формула `type1_triple` кодирует то, что все вершины, кроме первой одинакового цвета, при этом этот цвет отличен от цвета первой вершины. Формула `type2_triple` кодирует то, что цвет первой вершины отличен от цвета остальных вершин, а среди остальных есть две одинакового цвета, который отличен от цвета оставшейся вершины. Формула `type3_triple` кодирует случай, когда все 4 вершины имеют различный цвет.

Теорема `my_Triple_Coloring` утверждает, что любая правильная раскраска графа T является раскраской одного из этих типов. Ее доказательство включает в себя перебор всех возможных раскрасок, однако благодаря использованию языка `Ltac` и `goal matching` можно переиспользовать куски доказательства в ситуациях, отличающихся только перестановкой цветов или изоморфизма графа.

Полный код доказательства приведен в файле `my_Triple_Coloring.v` 4.4. Тактика `contr` доказывает `my_Triple_Coloring` от противного и применяется в случаях, когда раскраска не является правильной, т. е. существует пара смежных ребер одного цвета. Тактика `type1_tac` применяется, когда полученная раскраска является раскраской первого типа, тактики `type1_tac_left`, `type1_tac_middle`, `type1_tac_right` – раскраской второго типа (различны раскраски на различные случаи, какая пара вершин является парой одного цвета) и тактика `type3_tac` применяется для доказательства того, что раскраска является раскраской третьего типа.

Таким образом, для любой раскраски графа T существует тактика, с помощью которой можно доказать утверждение о том, что если раскраска правильная, то она является раскраской одной из трех типов. Теперь все эти тактики можно объединить в одну тактику `level4`, которая с помощью `goal matching` может определить, какую именно тактику из указанных использовать. Теперь можно создать тактики `level3` и `level2`, которые также с помощью `goal matching` определяют, необходимо ли доказывать утверждение от противного или вызывать тактику следующего уровня.

Итак, благодаря `goal matching` доказательство утверждения при различных контекстах может быть доказано одной и той же тактикой, что позволяет использовать конвейер и записать доказательство теоремы очень кратко

Lemma coloring_triple_T:

```
forall c: Coloring, is_good_coloring c T ->
type1_triple [1; 2; 3; 4] c \/ type2_triple [1; 2; 3; 4] c \/
type3_triple [1; 2; 3; 4] c.
```

Proof.

```
intros. unfold is_good_coloring in H. unfold is_coloring in H.
destruct H. remember H as H'. clear HeqH'.
specialize (H' 1). inversion H';
remember H as H''; clear HeqH''; specialize (H'' 2);
inversion H''; remember H0 as H0'; clear HeqH0';
level2 H H0' H0 H2 H3 c.
```

Qed.

2.3 Типы возможных правильных раскрасок графа H в не более чем 4 цвета

Теперь, когда доказано утверждение про правильные раскраски *троек*, можно перейти к раскраскам графа H .

Типы раскрасок графа H в не более чем 4 цвета можно описать следующими функциями в системе Coq.

(* Type1 triple - Type1 triple *)

Definition type1_H (c: Coloring) : Prop :=

```
type1_triple [1; 2; 4; 6] c /\ type1_triple [1; 3; 5; 7] c /\
~ same_color c 2 3.
```

(* Type1 triple - Type2 triple *)

Definition type2_H (c: Coloring) : Prop :=

```
(type1_triple [1; 2; 4; 6] c /\ type2_triple [1; 3; 5; 7] c /\
~ same_color c 2 3 /\ ~ same_color c 2 5 /\ ~ same_color c 2 7) \/
(type2_triple [1; 2; 4; 6] c /\ type1_triple [1; 3; 5; 7] c /\
~ same_color c 2 3 /\ ~ same_color c 4 3 /\ ~ same_color c 6 3).
```

```

(* Diagonals are monochromatic *)
Definition type3_H (c: Coloring) : Prop :=
  type3_triple [1; 2; 4; 6] c /\ type3_triple [1; 3; 5; 7] c /\
    same_color c 2 5 /\ same_color c 3 6 /\ same_color c 4 7.

(* One monochromatic diagonal and
same colors close to the vertices in diagonal *)
Definition type4_H (c: Coloring) : Prop :=
  type2_triple [1; 2; 4; 6] c /\ type2_triple [1; 3; 5; 7] c /\
  (
    (* Diagonal is 2 5 *)
    (same_color c 2 5 /\ same_color c 3 7 /\ same_color c 4 6 ) \/

    (* Diagonal is 3 6 *)
    (same_color c 3 6 /\ same_color c 2 4 /\ same_color c 5 7 ) \/

    (* Diagonal is 4 7 *)
    (same_color c 4 7 /\ same_color c 3 5 /\ same_color c 2 6 )
  ).

```

В системе Coq теорема о возможных раскрасках графа H формулируется следующим образом:

```

Lemma coloring_triple:
  forall c: Coloring, is_good_coloring c H ->
  type1_H c \/ type2_H c \/ type3_H c \/ type4_H c.

```

Полный код доказательства можно найти в файле `my_H_Coloring.v` 4.5.

Также как и в прошлой секции, можно определить тактику `contr`, которая доказывает утверждение от противного, т. е. доказывает, что данная раскраска не является правильной. Но так как теперь вершин в графе больше и не всегда понятно, какие именно смежные вершины окрашены в одинаковый цвет, доказательство упрощает использование тактики `find_contr`, которая перебирает гипотезы-посылки о том, что какие-то две вершины покрашены в один цвет, в контексте доказательства и пытается из этих гипотез вывести, что раскраска не является правильной.

Также определены тактики

`type1_H_tac`, `type2_H_tac_left_left`, `type2_H_tac_left_right`,
`type2_H_tac_left_middle`, `type2_H_tac_right_left`,
`type2_H_tac_right_middle`, `type2_H_tac_right_right`, `type3_H_tac`,
`type4_H_tac_1`, `type4_H_tac_2`, `type4_H_tac_3` для каждого подтипа рас-
 краски. Для автоматизации использования этих тактик разработана еще
 одна тактика `find_type`, в которой используется оператор соответствия цели
 (goal matching). Тактика `color_next` «окрашивает следующую вершину»,
 т. е. выводит посылку о том, что цвет следующей вершины принадлежит
 индуктивному типу `is_color` и делает инверсию этого типа.

Итоговое доказательство теоремы выглядит следующим образом:

Lemma coloring_H:

```
forall c: Coloring, is_good_coloring c H ->
  type1_H c \/ type2_H c \/ type3_H c \/ type4_H c.
```

Proof.

```
intros. unfold is_good_coloring in H.
unfold is_coloring in H. destruct H.
color_next H 1;
  color_next H 2; try find_contr H0 c;
  color_next H 3; try find_contr H0 c;
  color_next H 4; try find_contr H0 c;
  color_next H 5; try find_contr H0 c;
  color_next H 6; try find_contr H0 c;
  color_next H 7; try find_contr H0 c;
  find_type H3 H5 H7 H9 H11 H13 H15 c.
```

Qed.

Глава 3. Заключение

3.1 Выводы

В данной работы были разработаны методы построения и графов, а также верифицирована корректность операций над ними. Формализованы и верифицированы утверждения о том, что есть не более 4 существенно различных способа раскрасить граф H в не более, чем 4 цвета, а также алгоритм перебора возможных раскрасок графа, использующий алгоритм возврата.

3.2 Планы будущей работы

Разработанные методы можно использовать для верификации других статей про раскраски графов, например, статьи Marijn J.H. Heule, «Computing Small Unit-Distance Graphs with Chromatic Number 5» [5].

Также разработанную технику можно использовать в автоматизации поиска графов меньших размеров, которые не красятся в 4 цвета, а также поиска графов, которые не красятся в 5 цветов.

Глава 4. Приложение

Листинг 4.1 подгружается из внешнего файла.

Листинг 4.1

Листинг myGraphs.v

```

From VFA Require Import Perm.
From VFA Require Import Color.

5 Open Scope positive.

(*
Definition add_edge (e: (E.t*E.t)) (g: graph) : graph :=
  M.add (fst e) (S.add (snd e) (adj g (fst e)))
10 (M.add (snd e) (S.add (fst e) (adj g (snd e)))) g).
*)
Definition add_edges (el: list (E.t*E.t)) (g: graph) : graph :=
  fold_right add_edge g el.

15 Definition mk_graph (el: list (E.t*E.t)) :=
  fold_right add_edge (M.empty _) el.

Definition G :=
  mk_graph [ (5,6); (6,2); (5,2); (1,5); (1,2); (2,4); (1,4)].
20

Compute (S.elements (Mdomain G)). (* = [4; 2; 6; 1; 5] *)

Definition K3 :=
25 mk_graph [ (1, 2) ; (2, 3); (1, 3)].

Compute (S.elements (Mdomain K3)).

Fixpoint l_rng' (l : list node) (cur_min: node) (cur_max: node)
  : node * node :=
30 match l with
| nil => (cur_min, cur_max)
| x :: xs => let cur_min := if x <? cur_min then x else
  cur_min in

```

```

        let cur_max := if cur_max <? x then x else
            cur_max in
        l_rng' (xs) (cur_min) (cur_max)
35 end.

Function l_rng (l : list node) :=
    match l with
    | nil => (1%positive, 2%positive)
40 | x::xs => l_rng' l x x
    end.

Function gr_rng (g : graph) : node * node :=
    l_rng (S.elements (Mdomain g)).
45

Compute gr_rng G.

Check M.add.

50 Definition rename_node (old : node) (new : node) (g : graph) :
    graph :=
    let nigh := adj g old in
    S.fold (fun n g' => add_edge (new, n) g') nigh (remove_node
        old g).

Function gr_show (g : graph) : list (node * node) :=
55 S.fold (fun n l => (map (fun y => (n, y)) (S.elements (adj g n
    ))) ++ l) (Mdomain g) nil.

Compute gr_show K3.

60 Compute gr_show (rename_node 3 1 (rename_node 2 7 (rename_node 1
    5 K3))).
    Compute S.elements (Mdomain (rename_node 3 1 (rename_node 2 7 (
        rename_node 1 5 K3)))).

(* The user should avoid any collision of the new and old names.
    *)

Function rename_all (f : node -> node) (g : graph) : graph :=
65 S.fold (fun n g' => rename_node n (f n) g') (Mdomain g) g.

```

```

Compute K3.
Compute gr_show (rename_all (fun x => x * 4) K3).

70 (* Connect two graphs by an articulation point (aka "sharnir").
    That point MUST be present in both graphs. *)

Compute gr_rng K3.

75 (* Deletes one instance, if it's present, otherwise doesn't
    change the list *)
Fixpoint delete_from_list (l: list node) (n: node) : list node
:=
  match l with
  | nil => nil
  | h::xs => if h =? n
80           then xs
             else h::(delete_from_list xs n)
  end.

Compute delete_from_list [4; 3; 1; 2; 5] 6.

85 (* n == len(before) *)
Fixpoint sort (n: nat) (before: list node) (after: list node) :
  list node :=
  match n with
  | 0 => after
90   | S n' =>
      let min_value := fst (l_rng before) in
      let before' := delete_from_list before min_value in
      sort n' before' ( after ++ [min_value] )
  end.

95
Definition rename_in_order (g: graph) : graph :=
  let sorted_vertices := sort (length (S.elements (Mdomain g)))
    (S.elements (Mdomain g)) nil in
  fst ( fold_left
100      (fun pair_g_next n =>
          let next_node := snd pair_g_next in
          let g' := fst pair_g_next in
          (

```



```

105         rename_node n next_node g',
            next_node+1
        )
    )
    sorted_vertices (g, 1)
).

110

Definition mk_art (g1 g2 : graph) (n m : node) : graph :=
    let g2' := rename_all (fun x => x + snd(gr_rng g1)) g2 in
    let m' := m + snd(gr_rng g1) in
115    let g := S.fold (fun m g' => M.add m (adj g2' m) g') (Mdomain
        g2') g1 in
    rename_node m' n g.

Compute gr_show (mk_art K3 K3 1 2).
Compute S.elements (Mdomain (mk_art K3 K3 1 1)).

120

Definition delete_edge (g: graph) (a b : node) : graph :=
    let a_neigh := S.remove b (adj g a) in
    let b_neigh := S.remove a (adj g b) in
125    M.add b b_neigh (M.add a a_neigh g).

Definition mk_cmn_edge (g1 g2 : graph) (a b n m : node) : graph
:=
(* Make graphs disjoint. *)
130    let g2' := rename_all (fun x => x + snd (gr_rng g1)) g2 in
(* New names for the edge's vertices. *)
    let n' := n + snd (gr_rng g1) in
    let m' := m + snd (gr_rng g1) in
(* Delete adge from second graph *)
135    let g2' := delete_edge g2' n' m' in
    let g_result := S.fold (fun m g' => M.add m (adj g2' m) g') (
        Mdomain g2') g1 in
    let g_result := rename_node n' a g_result in
    rename_node m' b g_result.

140 (* articulation by 2 non adjacent points in one graph to build J
    *)

```

```

Compute gr_show (mk_cmn_edge K3 K3 1 3 1 3).
Compute gr_show (rename_in_order (mk_cmn_edge K3 K3 1 3 1 3)).

145 (* Make graph H. *)
Definition H : graph :=
  let g1 := rename_in_order (mk_cmn_edge K3 K3 1 3 1 3) in
  let g2 := rename_in_order (mk_cmn_edge g1 K3 1 (snd (gr_rng g1
    )) 1 3) in
150 let g3 := rename_in_order (mk_cmn_edge g2 K3 1 (snd (gr_rng g2
    )) 1 3) in
  let g4 := rename_in_order (mk_cmn_edge g3 K3 1 (snd (gr_rng g3
    )) 1 3) in
  rename_in_order (add_edge (2, snd (gr_rng g4)) g4).

Compute gr_show H.

155
Definition J: graph :=
  let HH := mk_cmn_edge H H 2 3 6 7 in
  let HH_H := mk_cmn_edge HH H 7 2 6 7 in
160 let HHH := rename_node 14 12 HH_H in
  let HHH_H := mk_cmn_edge HHH H 6 7 6 7 in
  let HHHH := rename_node 19 17 HHH_H in
  let HHHH_H := mk_cmn_edge HHHH H 5 6 6 7 in
  let HHHHH := rename_node 24 22 HHHH_H in
165 let HHHHH_H := mk_cmn_edge HHHHH H 4 5 6 7 in
  let HHHHHH := rename_node 29 27 HHHHH_H in
  let HHHHHH_H := mk_cmn_edge HHHHHH H 3 4 6 7 in
  let HHHHHHH := rename_node 34 32 HHHHHH_H in
  rename_in_order (rename_node 37 9 HHHHHHH).

170

(* Centers: 1, 8, 13, 17, 21, 25, 29 *)
(* Linking vertices: 9, 12, 16, 20, 24, 28 *)

175 Compute gr_show J.

Definition K: graph :=

```

```

180   let JJ := mk_art J J 1 1 in
      let JJ := add_edges [(9, 9+31); (12, 12+31); (16, 16+31); (20,
        20+31);
                          (24, 24+31); (28, 28+31)] JJ in
      rename_in_order JJ.

Compute gr_show K.

185

Definition A := 9.
Definition B := 20.

190 Definition L: graph :=
      let KK := mk_art K K A A in
      let KK := add_edge (B, B+snd(gr_rng K)) KK in
      rename_in_order KK.

195 Compute gr_show L.

Close Scope positive.

```

Листинг 4.2

Листинг myGraphs_Properties.v

```

From VFA Require Import myGraphs.
From VFA Require Import Color.
From VFA Require Import Perm.

5
Open Scope positive.

Definition graph_ok (g : graph) :=
  undirected g /\ no_selfloop g.

10
Definition gr_deg (g : graph) (n : node) : nat :=
  S.cardinal (adj g n).

Definition edgeb (g : graph) (n m : node) :=
15  S.mem n (adj g m).

Definition edge (g : graph) (n m : node) :=
  S.In n (adj g m).

```

```

20 Lemma adj_M_In : forall g n m,
    S.In m (adj g n) -> M.In n g.
Proof. intros. unfold adj in H.
destruct (M.find n g) eqn: H1.
- apply M.find_2 in H1. Print M.In.
25   exists n0. assumption.
- discriminate.
Qed.

Check M.fold.

30
(* Dual to Mdomain and nodes. *)
Definition conodes (g: graph) : nodeset :=
    M.fold (fun _ a s => S.union a s) g S.empty.
(* Let's try to avoid using this. *)

35
(* Of course, for an undirected graph g, nodes g = conodes g. *)
Compute S.elements (nodes H).
Compute S.elements (conodes H).

40 Lemma edge_sym : forall g n m, graph_ok g ->
    edge g n m -> edge g m n.
Proof.
  intros. unfold graph_ok, undirected in H. destruct H as [H _].
  unfold edge. apply H. assumption.
45 Qed.

Lemma edge_irrefl : forall g n, graph_ok g -> ~ edge g n n.
Proof.
  intros. unfold graph_ok, no_selfloop in H. destruct H as [_ H].
50 unfold edge. apply H.
  Qed.

(* The weak versions independent of symmetry *)
Lemma edge_corr_1 : forall g n m, edge g n m -> S.In m (nodes g)
.

55 Proof.
  intros. unfold nodes. rewrite Sin_domain.
  apply adj_M_In with n. unfold edge in H. assumption.
  Qed.

```

```

60 (*
    Lemma edge_corr_2 : forall g n m, edge g n m -> S.In n (conodes
      g).
    Proof.
      intros. unfold conodes. Search M.fold.
      Admitted.
65 (* Let's try to avoid using this. *)
    *)

    Lemma edge_corr : forall g n m, graph_ok g ->
      edge g n m -> S.In n (nodes g) /\ S.In m (nodes g).
70 Proof.
    intros; split; [ apply edge_sym in H0 | idtac ];
    [ apply edge_corr_1 with m | idtac | apply edge_corr_1 with n];
      assumption.
    Qed.

75 (* Our graphs K3, H, J, K, L are graphs indeed. *)

    (* All these facts can be established by a direct computation.
      But we HAVE TO bound the quantifiers on graph nodes.
    *)
80
    Require Export List.
    Require Export Sorted.
    Require Export Setoid Basics Morphisms.

85 Lemma K3_ok : graph_ok K3.
    Proof. split.
      - unfold undirected. intros. remember H as H'.
        clear HeqH'. apply edge_corr_1 in H.
        (* Here lies the truth! *)
90 Ltac gr_destr h := apply S.elements_1 in h; compute in h;
      repeat rewrite InA_cons in h; rewrite InA_nil in h;
      repeat destruct h as [? | h]; try inversion h; subst.
      gr_destr H; gr_destr H'; reflexivity.

95 - unfold no_selfloop. repeat intro. remember H as H'.
      clear HeqH'. apply edge_corr_1 in H. gr_destr H; gr_destr H';
      discriminate.

```

```

Qed.

100 Lemma H_ok : graph_ok H.
    Proof.
      split.
      - unfold undirected. intros. remember H as H'.
        clear HeqH'. apply edge_corr_1 in H.
105   gr_destr H; gr_destr H'; reflexivity.
      - unfold no_selfloop. repeat intro. remember H as H'.
        clear HeqH'. apply edge_corr_1 in H. gr_destr H; gr_destr H';
        discriminate.
    Qed.

110 Lemma J_ok : graph_ok J.
    Proof.
      split.
      - unfold undirected. intros. remember H as H'.
115   clear HeqH'. apply edge_corr_1 in H.
        gr_destr H; gr_destr H'; reflexivity.
      - unfold no_selfloop. repeat intro. remember H as H'.
        clear HeqH'. apply edge_corr_1 in H. gr_destr H; gr_destr H';
        discriminate.

120 Qed.

    Check J_ok.

    Lemma K_ok : graph_ok K.
125 Proof.
      split.
      - unfold undirected. intros. remember H as H'.
        clear HeqH'. apply edge_corr_1 in H.
        gr_destr H; gr_destr H'; reflexivity.
130 - unfold no_selfloop. repeat intro. remember H as H'.
        clear HeqH'. apply edge_corr_1 in H. gr_destr H; gr_destr H';
        discriminate.
    Qed.

135 Lemma L_ok : graph_ok L.
    Proof.
      split.

```

```

- unfold undirected. intros. remember H as H'.
140 clear HeqH'. apply edge_corr_1 in H.
    gr_destr H; gr_destr H'; reflexivity.
- unfold no_selfloop. repeat intro. remember H as H'.
    clear HeqH'. apply edge_corr_1 in H. gr_destr H; gr_destr H';
    discriminate.
145 Qed.

Lemma add_edge_corr' : forall g x y a b,
    edge (add_edge (a, b) g) x y <-> edge g x y \/ (x = a /\ y = b
    ) \/ (x = b /\ y = a).
150 Proof.
    intros. pattern g. remember (fun g0 : graph =>
    edge (add_edge (a, b) g0) x y <-> edge g0 x y \/ (x = a /\ y =
    b) \/ (x = b /\ y = a)) as P.
    apply WP.map_induction; intros.
    - rewrite HeqP. unfold add_edge, edge. simpl.
155 rewrite M.Empty_alt in H. unfold adj. repeat rewrite H.
    repeat rewrite WF.add_o. assert (H1 : S.In x S.empty <-> False
    ).
    { split. apply Snot_in_empty. tauto. } destruct (WP.F.eq_dec
    a y).
    + rewrite S.add_spec, e. rewrite H1. split; intro.
      * destruct H0; subst; tauto.
160 * repeat destruct H0 as [ ? | H0]; try contradiction;
      left; destruct H0; rewrite H0; try rewrite H2; reflexivity
      .
    + destruct (WP.F.eq_dec b y).
      * rewrite S.add_spec, e. rewrite H1. split; intro.
        { destruct H0; subst; tauto. }
165 { repeat destruct H0 as [ ? | H0]; try contradiction;
        left; destruct H0; rewrite H0; try rewrite H2; reflexivity
        . }
      * rewrite H, H1. split; intro; try contradiction. repeat
        destruct H0 as [? | H0];
        [ assumption | destruct n0 | destruct n ]; symmetry; tauto
        .
    - rewrite HeqP in *. clear HeqP. unfold WP.Add in H1. unfold
      add_edge, edge in *.
170 simpl in *. unfold adj in *. repeat rewrite H1, WF.add_o in *.

```

```

destruct (WP.F.eq_dec a y); repeat rewrite WF.add_o in *.
+ destruct (WP.F.eq_dec x0 a), (WP.F.eq_dec x0 y).
  * rewrite S.add_spec. split; intro H2; repeat destruct H2 as
    [? | H2].
    { repeat right. rewrite H2, <- e1, <- e2. tauto. } { tauto
      . }
175   { tauto. } { left. destruct H2. rewrite <-H3, H2, <-e2, <-
      e1. reflexivity. }
    { left. destruct H2. rewrite H2. reflexivity. }
  * rewrite e0 in *. contradiction.
  * rewrite e0 in *. contradiction.
  * rewrite e0 in *. rewrite <- H. reflexivity.
180 + destruct (WP.F.eq_dec b y), (WP.F.eq_dec x0 y).
  * destruct (WP.F.eq_dec x0 b).
    { rewrite S.add_spec. split; intro H2; repeat destruct H2
      as [? | H2];
      try tauto. { rewrite <-e1, <-e2, H2. tauto. } { destruct
        H2. rewrite H2. tauto. }
      { left. destruct H2. rewrite <-H3, <-e1, H2, <-e2. tauto
        . }
185   }
    { subst. contradiction. }
  * destruct (WP.F.eq_dec x0 b).
    { subst. contradiction. }
    { rewrite H. reflexivity. }
190 * split; try tauto. intro H2; repeat destruct H2 as [? | H2
    ]; try tauto;
    destruct H2; subst; contradiction.
  * exact H.
Qed.

195 Lemma add_edge_corr : forall g a b, graph_ok g -> a <> b ->
    graph_ok (add_edge (a, b) g).
Proof.
unfold graph_ok. intros. split.
- unfold undirected. intros. apply add_edge_corr'. apply
  add_edge_corr' in H1.
200 repeat destruct H1 as [? | H1].
  + left. apply edge_sym; assumption.
  + tauto.
  + tauto.

```



```

- unfold no_selfloop. repeat intro. apply add_edge_corr' in H1.
205   repeat destruct H1 as [? | H1].
      + apply edge_irrefl with (g := g) (n := i); assumption.
      + destruct H1. subst. contradiction.
      + destruct H1. subst. contradiction.
Qed.

210 Lemma pos_eq_dec : forall x y : S.elts, x = y \/ x <> y.
Proof.
  intros; destruct (Pos.lt_total x y) as [? | [? | ?]];
  try (right; intro; rewrite H0 in H; destruct (Pos.lt_irrefl y);
215     assumption); tauto.
Qed.

(* Monochromatic triplet in H with center. *)

220 Definition center (g : graph) (o : node) : Prop :=
  forall i, S.In i (nodes g) -> i <> o -> edge g i o.

Definition H_center (o : node) : Prop :=
225   (gr_deg H o) = 6%nat.

Compute (gr_deg H 1).

230 Check subset_nodes.

Definition gr_deg_search (g : graph) (d : nat) : nodeset :=
  subset_nodes (fun _ a => Nat.eqb (S.cardinal a) d) g.

235 Compute S.elements (gr_deg_search H 0).

Fixpoint gr_deg_sort (g : graph) (maxd : nat) : list (list node)
  :=
  match maxd with
  | 0%nat => [S.elements (gr_deg_search g 0)]
240   | S n => S.elements (gr_deg_search g maxd) :: gr_deg_sort g n
  end.

Compute gr_deg_sort H 6.

```

```

245 Definition node_color (clr : coloring) (n : node) (c : S.elc) :=
    M.find n clr = Some c.

250 Definition monochrom (g : graph) (clr : coloring) (o l m n :
    node) :=
    edge g o l /\ edge g o m /\ edge g o n /\
    exists c, (node_color clr l c /\ node_color clr m c /\
    node_color clr n c).

255 Lemma H_monochrom_center : forall (plt : S.t) (clr : coloring) (
    o l m n : node),
    coloring_ok plt H clr -> monochrom H clr o l m n -> H_center o
    .

260 Definition palette4: S.t := fold_right S.add S.empty [1; 2; 3;
    4].

    Compute (M.elements (color palette H)).

265 Close Scope positive.

```

Листинг 4.3

Листинг my_New_Coloring.v

```

Require Export Coq.Bool.Bool.
Require Export Coq.Lists.List.
Require Export Coq.Lists.ListSet.
5 Require Export Coq.Numbers.BinNums.
Export ListNotations.
From VFA Require Import Color.

Definition Coloring := positive -> positive.
10 Definition same_color (c : Coloring) (u v : positive) : Prop :=
    c u = c v.

```

```

Inductive is_color : positive -> Prop :=
| c1: is_color 1%positive
| c2: is_color 2%positive
15 | c3: is_color 3%positive
| c4: is_color 4%positive
.

Definition is_coloring (c : Coloring) := forall x : positive,
    is_color (c x).
20

Definition is_good_coloring (c : Coloring) (g : graph) :=
    is_coloring c /\ forall x y : positive, S.In y (adj g x) -> c
        x <> c y.

Definition is_colorable (g : graph) :=
25 exists c : Coloring, is_good_coloring c g.

```

Листинг 4.4

Листинг my_Triple_Coloring.v

```

From VFA Require Import Color.
From VFA Require Import Perm.
From VFA Require Import myGraphs.
5 From VFA Require Import Graphs_Properties.
From VFA Require Import myColoringSmallGraphs.
From VFA Require Import my_New_Coloring.

Open Scope positive.
10

Compute nth 0 [1;2;3] 1.

(* Monochromatic *)
15 Definition type1_triple (el: list node) (c: Coloring) :=
    let center := nth 0 el 1 in
    let v1 := nth 1 el 1 in
    let v2 := nth 2 el 1 in
    let v3 := nth 3 el 1 in
20 let c1 := c center in
    let c2 := c v1 in
    ~ (c1 = c2) /\ same_color c v1 v2 /\ same_color c v2 v3.

```

```

(* 2 and 1 *)
25 Definition type2_triple (el: list node) (c: Coloring) :=
    let center := nth 0 el 1 in
    let v1 := nth 1 el 1 in
    let v2 := nth 2 el 1 in
    let v3 := nth 3 el 1 in
30
    let c1 := c center in
    let c2 := c v1 in
    let c3 := c v2 in
    let c4 := c v3 in
35 ~ (c1 = c2) /\ ~ (c1 = c3) /\ ~ (c1 = c4) /\
    ( (c2 = c3 /\ ~ c2 = c4) \/ (c2 = c4 /\ ~ c2 = c3) \/ (c3 =
        c4 /\ ~ c3 = c2) ).

Definition type3_triple (el: list node) (c: Coloring) :=
    let center := nth 0 el 1 in
40 let v1 := nth 1 el 1 in
    let v2 := nth 2 el 1 in
    let v3 := nth 3 el 1 in

    let c1 := c center in
45 let c2 := c v1 in
    let c3 := c v2 in
    let c4 := c v3 in
    ~ (c1 = c2) /\ ~ (c1 = c3) /\ ~ (c1 = c4) /\
    (~ c2 = c3) /\ (~ c2 = c4) /\ (~ c3 = c4).
50

Ltac type2_tac_left h2 h3 h4 h5 := right; left; unfold
    type2_triple;
    simpl; rewrite <- h2; rewrite <- h3; rewrite <- h4; rewrite
        <- h5;
    repeat split; cbv; try intro; try inversion H1;
55 left; repeat split; cbv; try intro; simpl; try inversion H1.

Ltac type2_tac_middle h2 h3 h4 h5 := right; left; unfold
    type2_triple;
    simpl; rewrite <- h2; rewrite <- h3; rewrite <- h4; rewrite
        <- h5;

```

```

repeat split; cbv; try intro; try inversion H1;
60 right; left; repeat split; cbv; try intro; simpl; try
    inversion H1.

Ltac type2_tac_right h2 h3 h4 h5 := right; left; unfold
    type2_triple;
    simpl; rewrite <- h2; rewrite <- h3; rewrite <- h4; rewrite
        <- h5;
    repeat split; cbv; try intro; try inversion H1;
65 right; right; repeat split; cbv; try intro; simpl; try
    inversion H1.

Ltac type3_tac h2 h3 h4 h5 := right; right; unfold type3_triple;
    simpl; rewrite <- h2; rewrite <- h3; rewrite <- h4; rewrite
        <- h5;
    repeat split; cbv; try intro; try inversion H1.
70

Ltac type1_tac h2 h3 h4 h5 := left; unfold type1_triple; split;
    simpl; try rewrite <- h2; try rewrite <- h3; cbv;
    try intro; try inversion H1;
    unfold same_color; try rewrite <- h3; try rewrite <- h4; try
        rewrite <- h5;
75 split; reflexivity.

Ltac contr h0' h2 h3 h4 h5 c n x :=
    let H1 := fresh in
    let H6 := fresh in
80 specialize (h0' 1 n);
    rewrite <- h5 in h0'; rewrite <- h2 in h0'; exfalse;
    assert (x <> x -> False);
    [> cbv; try intro; assert (x =x);
    try reflexivity; apply H1; apply H6 |
85 apply H1; apply h0'; reflexivity ].

Ltac level4 h0' h2 h3 h4 h5 c :=
    match goal with
    | [H2 : ?x = c 1, Hn : ?x = c ?n |- type1_triple _ _ \/  

        type2_triple _ _ \/  

        type3_triple _ _] =>
90 contr h0' h2 h3 h4 h5 c n x
    (*let H1 := fresh in
        let H6 := fresh in

```

```

specialize (h0' 1 n);
rewrite <- h5 in h0'; rewrite <- h2 in h0'; exfalso;
95  assert (x <> x -> False);
    [> cbv; try intro; assert (x =x);
    try reflexivity; apply H1; apply H6 |
      apply H1; apply h0'; reflexivity ] *)
| [ H3 : ?x = c 2, H4 : ?x = c 3, H5 : ?x = c 4 |-
  type1_triple _ _ \/ type2_triple _ _ \/ type3_triple _ _]
=>
100  type1_tac h2 h3 h4 h5
| [ H3 : ?x = c 2, H5 : ?x = c 4 |- type1_triple _ _ \/
  type2_triple _ _ \/ type3_triple _ _] =>
  type2_tac_middle h2 h3 h4 h5
| [ H4 : ?x = c 3, H5 : ?x = c 4 |- type1_triple _ _ \/
  type2_triple _ _ \/ type3_triple _ _] =>
  type2_tac_right h2 h3 h4 h5
105 | [ H3 : ?x = c 2, H4 : ?x = c 3 |- type1_triple _ _ \/
  type2_triple _ _ \/ type3_triple _ _] =>
  type2_tac_left h2 h3 h4 h5
| [ H2 : ?x = c 1, H3 : ?y = c 2, H4 : ?z = c 3, H5 : ?w = c 4
  |- type1_triple _ _ \/ type2_triple _ _ \/ type3_triple _
  _] =>
  type3_tac h2 h3 h4 h5
end.
110

Ltac level3 h h0' h0 h2 h3 h4 c :=
  match goal with
  | [ H2 : ?x = c 1, H4 : ?x = c 3 |- type1_triple _ _ \/
    type2_triple _ _ \/ type3_triple _ _] =>
115  let Ha := fresh in
    specialize (h0' 1 3);
    rewrite <- h4 in h0'; rewrite <- h2 in h0'; exfalso;
    assert (x <> x -> False);
    [> cbv; intro Ha; assert (x = x) as H5 ;
120    [> try reflexivity |
      apply Ha in H5; apply H5 ]
    | apply Ha; apply h0'; reflexivity ]
| [ |- type1_triple _ _ \/ type2_triple _ _ \/ type3_triple _
  _] =>

```

```

remember h as H'''' eqn:HeqH'''' ; clear HeqH'''';
specialize (H'''' (3+1)); inversion H'''' as [H5|H5|H5|H5
];
125 remember h0 as h0'' eqn:HeqH0' ; clear HeqH0';
level4 h0'' h2 h3 h4 H5 c
end.

Ltac level2 h h0' h0 h2 h3 c :=
130 match goal with
| [ H2 : ?x = c 1, H4 : ?x = c 2 |- type1_triple _ _ \/  
type2_triple _ _ \/  
type3_triple _ _ ] =>
let Ha := fresh in
specialize (h0' 1 2);
rewrite <- h3 in h0'; rewrite <- h2 in h0'; exfalso;
135 assert (x <> x -> False);
[> cbv; intro Ha; assert (x = x) as H5 ;
[> try reflexivity |
apply Ha in H5; apply H5 ]
| apply Ha; apply h0'; reflexivity ]
140 | [ |- type1_triple _ _ \/  
type2_triple _ _ \/  
type3_triple _ _ ] =>
remember h as H''' eqn:HeqH''' ; clear HeqH'''; specialize (
H''' (2+1)); inversion H''' as [H4|H4|H4|H4];
remember h0 as Ha eqn:HeqH0 ; clear HeqH0;
level3 h Ha h0 h2 h3 H4 c
end.
145

Definition T := mk_graph [ (1, 2); (1, 3); (1, 4) ].

Lemma coloring_triple_T:
150 forall c: Coloring, is_good_coloring c T ->
type1_triple [1; 2; 3; 4] c \/  
type2_triple [1; 2; 3; 4] c \/  
type3_triple [1; 2; 3; 4] c.
Proof.
intros. unfold is_good_coloring in H. unfold my_New_Coloring.
is_coloring in H. destruct H.
remember H as H'. clear HeqH'. specialize (H' 1). inversion H
';
155 remember H as H''; clear HeqH''; specialize (H'' 2);
inversion H''; remember H0 as H0'; clear HeqH0';

```

```

    level2 H H0' H0 H2 H3 c.
Qed.

160 Close Scope positive.

```

Листинг 4.5

Листинг my_H_coloring.v

```

From VFA Require Import Color.
From VFA Require Import Perm.
From VFA Require Import myGraphs.
5 From VFA Require Import Graphs_Properties.
From VFA Require Import myColoringSmallGraphs.
From VFA Require Import my_New_Coloring.
From VFA Require Import my_Triple_Coloring.

10 Open Scope positive.

(* Type1 - Type1 *)
Definition type1_H (c: Coloring) : Prop :=
15   type1_triple [1; 2; 4; 6] c /\ type1_triple [1; 3; 5; 7] c /\
    ~ same_color c 2 3.

(* Type1 - Type2 *)
Definition type2_H (c: Coloring) : Prop :=
20   (type1_triple [1; 2; 4; 6] c /\ type2_triple [1; 3; 5; 7] c /\
    ~ same_color c 2 3 /\ ~ same_color c 2 5 /\ ~ same_color c 2
    7) \/
    (type2_triple [1; 2; 4; 6] c /\ type1_triple [1; 3; 5; 7] c /\
    ~ same_color c 2 3 /\ ~ same_color c 4 3 /\ ~ same_color c 6
    3).

25 (* Diagonals are monochromatic *)
Definition type3_H (c: Coloring) : Prop :=
    type3_triple [1; 2; 4; 6] c /\ type3_triple [1; 3; 5; 7] c /\
    same_color c 2 5 /\ same_color c 3 6 /\ same_color c 4 7.

30

(* One diagonal and same colors close to the vert in diagonal *)
Definition type4_H (c: Coloring) : Prop :=

```



```

type2_triple [1; 2; 4; 6] c /\ type2_triple [1; 3; 5; 7] c /\
(
35   (* Diagonal is 2 5 *)
      (same_color c 2 5 /\ same_color c 3 7 /\ same_color c 4 6 )
      \/

      (* Diagonal is 3 6 *)
      (same_color c 3 6 /\ same_color c 2 4 /\ same_color c 5 7 )
      \/
40   (* Diagonal is 4 7 *)
      (same_color c 4 7 /\ same_color c 3 5 /\ same_color c 2 6 )
    ).

45 Ltac contr H0 Hn Hm n m x :=
    let H1 := fresh in
    let H6 := fresh in
    remember H0 as H0' eqn:HeqH0'; clear HeqH0';
    specialize (H0' n m);
50   rewrite <- Hn in H0'; rewrite <- Hm in H0'; exfalso;
    assert (x <> x -> False);
    [> cbv; try intro; assert (x =x);
    try reflexivity; apply H1; apply H6 |
      apply H1; apply H0'; reflexivity ].

55

Ltac find_contr H0 c :=
    lazy match goal with
    | [H2 : ?x = c 1, Hn : ?x = c ?n |- type1_H _ \/ type2_H _ \/
      type3_H _ \/ type4_H _] =>
60      contr H0 H2 Hn 1 n x
    | [Hn : ?x = c ?n, Hm : ?x = c ?m, Hk : ?x = c ?k |- type1_H _
      \/ type2_H _ \/ type3_H _ \/ type4_H _] =>
      try contr H0 Hn Hm n m x; try contr H0 Hn Hk n k x;
      try contr H0 Hm Hk m k x
    | [Hn : ?x = c ?n, Hm : ?x = c ?m |- type1_H _ \/ type2_H _ \/
      type3_H _ \/ type4_H _] =>
65      contr H0 Hn Hm n m x
    end.

Ltac color_next H n :=

```

```

    let H' := fresh in
70   let HeqH' := fresh in
      remember H as H' eqn: HeqH'; clear HeqH'; specialize (H' n);
      inversion H'.

Ltac use_color H3 H5 H7 H9 H11 H13 H15 :=
  try rewrite <- H3;
75   try rewrite <- H5;
      try rewrite <- H7;
      try rewrite <- H9;
      try rewrite <- H11;
      try rewrite <- H13;
80   try rewrite <- H15.

Ltac trivia_cases H3 H5 H7 H9 H11 H13 H15 :=
  repeat split; simpl; unfold same_color;
  use_color H3 H5 H7 H9 H11 H13 H15;
85   try discriminate; try reflexivity.

Ltac type1_H_tac H3 H5 H7 H9 H11 H13 H15 :=
  left; unfold type1_H;
  trivia_cases H3 H5 H7 H9 H11 H13 H15.
90

Ltac type2_H_tac_left_left H3 H5 H7 H9 H11 H13 H15 :=
  right; left; unfold type2_H;
  (* Choose types of triples *)
  left; trivia_cases H3 H5 H7 H9 H11 H13 H15;
95   (* Chose the different color in Type2 *)
      left; split; trivia_cases H3 H5 H7 H9 H11 H13 H15.

Ltac type2_H_tac_left_right H3 H5 H7 H9 H11 H13 H15 :=
  right; left; unfold type2_H;
100  (* Choose types of triples *)
      left; trivia_cases H3 H5 H7 H9 H11 H13 H15;
      (* Chose the different color in Type2 *)
      right; right; split; trivia_cases H3 H5 H7 H9 H11 H13 H15.

105 Ltac type2_H_tac_left_middle H3 H5 H7 H9 H11 H13 H15 :=
  right; left; unfold type2_H;
  (* Choose types of triples *)
  left; trivia_cases H3 H5 H7 H9 H11 H13 H15;

```

```

(* Chose the different color in Type2 *)
110   right; left; split; trivia_cases H3 H5 H7 H9 H11 H13 H15.

Ltac type2_H_tac_right_left H3 H5 H7 H9 H11 H13 H15 :=
  right; left; unfold type2_H;
  right; trivia_cases H3 H5 H7 H9 H11 H13 H15;
115   left; split; trivia_cases H3 H5 H7 H9 H11 H13 H15.

Ltac type2_H_tac_right_middle H3 H5 H7 H9 H11 H13 H15 :=
  right; left; unfold type2_H;
  right; trivia_cases H3 H5 H7 H9 H11 H13 H15;
120   right; left; split; trivia_cases H3 H5 H7 H9 H11 H13 H15.

Ltac type2_H_tac_right_right H3 H5 H7 H9 H11 H13 H15 :=
  right; left; unfold type2_H;
  right; trivia_cases H3 H5 H7 H9 H11 H13 H15;
125   repeat right; split; trivia_cases H3 H5 H7 H9 H11 H13 H15.

Ltac type3_H_tac H3 H5 H7 H9 H11 H13 H15 :=
  right; right; left; trivia_cases H3 H5 H7 H9 H11 H13 H15.

130 Ltac type4_H_tac_1 H3 H5 H7 H9 H11 H13 H15 :=
  repeat right; unfold type4_H;
  trivia_cases H3 H5 H7 H9 H11 H13 H15;
  [> repeat right; trivia_cases H3 H5 H7 H9 H11 H13 H15 |
    right; left; trivia_cases H3 H5 H7 H9 H11 H13 H15 |
135   left; trivia_cases H3 H5 H7 H9 H11 H13 H15
  ].

Ltac type4_H_tac_2 H3 H5 H7 H9 H11 H13 H15 :=
  repeat right; unfold type4_H;
140   trivia_cases H3 H5 H7 H9 H11 H13 H15;
  [> left; trivia_cases H3 H5 H7 H9 H11 H13 H15 |
    repeat right; trivia_cases H3 H5 H7 H9 H11 H13 H15 |
    right; left; trivia_cases H3 H5 H7 H9 H11 H13 H15
  ].

145 Ltac type4_H_tac_3 H3 H5 H7 H9 H11 H13 H15 :=
  repeat right; unfold type4_H;
  trivia_cases H3 H5 H7 H9 H11 H13 H15;
  [> right; left; trivia_cases H3 H5 H7 H9 H11 H13 H15 |

```

```

150   left; trivia_cases H3 H5 H7 H9 H11 H13 H15 |
      repeat right; trivia_cases H3 H5 H7 H9 H11 H13 H15
    ].

Ltac find_type H3 H5 H7 H9 H11 H13 H15 c :=
155   match goal with
      | [ H3 : ?x1 = c 1, H5 : ?x2 = c 2, H7 : ?x3 = c 3,
          H9 : ?x2 = c 4, H11 : ?x3 = c 5, H13 : ?x2 = c 6,
          H15 : ?x3 = c 7 |- type1_H _ \/ type2_H _ \/ type3_H _
            \/ type4_H _ ] =>
          type1_H_tac H3 H5 H7 H9 H11 H13 H15
160
      | [ H3 : ?x1 = c 1, H5 : ?x2 = c 2, H7 : ?x3 = c 3,
          H9 : ?x2 = c 4, H11 : ?x3 = c 5, H13 : ?x2 = c 6,
          H15 : ?x4 = c 7 |- type1_H _ \/ type2_H _ \/ type3_H _
            \/ type4_H _ ] =>
          type2_H_tac_left_left H3 H5 H7 H9 H11 H13 H15
165
      | [ H3 : ?x1 = c 1, H5 : ?x2 = c 2, H7 : ?x3 = c 3,
          H9 : ?x2 = c 4, H11 : ?x4 = c 5, H13 : ?x2 = c 6,
          H15 : ?x3 = c 7 |- type1_H _ \/ type2_H _ \/ type3_H _
            \/ type4_H _ ] =>
          type2_H_tac_left_middle H3 H5 H7 H9 H11 H13 H15
170
      | [ H3 : ?x1 = c 1, H5 : ?x2 = c 2, H7 : ?x3 = c 3,
          H9 : ?x2 = c 4, H11 : ?x4 = c 5, H13 : ?x2 = c 6,
          H15 : ?x4 = c 7 |- type1_H _ \/ type2_H _ \/ type3_H _
            \/ type4_H _ ] =>
          type2_H_tac_left_right H3 H5 H7 H9 H11 H13 H15

      | [ H3 : ?x1 = c 1, H5 : ?x2 = c 2, H7 : ?x3 = c 3,
175          H9 : ?x2 = c 4, H11 : ?x3 = c 5, H13 : ?x4 = c 6,
          H15 : ?x3 = c 7 |- type1_H _ \/ type2_H _ \/ type3_H _
            \/ type4_H _ ] =>
          type2_H_tac_right_left H3 H5 H7 H9 H11 H13 H15

      | [ H3: ?x1 = c 1, H5 : ?x2 = c 2, H7 : ?x3 = c 3,
          H9 : ?x4 = c 4, H11 : ?x3 = c 5, H13 : ?x2 = c 6,
180          H15 : ?x3 = c 7 |- type1_H _ \/ type2_H _ \/ type3_H _
            \/ type4_H _ ] =>
          type2_H_tac_right_middle H3 H5 H7 H9 H11 H13 H15

      | [ H3 : ?x1 = c 1, H5 : ?x2 = c 2, H7 : ?x3 = c 3,
          H9 : ?x4 = c 4, H11 : ?x3 = c 5, H13 : ?x4 = c 6,

```

```

H15 : ?x3 = c 7 |- type1_H _ \/ type2_H _ \/ type3_H _
      \/ type4_H _ ] =>
185   type2_H_tac_right_right H3 H5 H7 H9 H11 H13 H15

| [ H3 : ?x1 = c 1, H5 : ?x2 = c 2, H7 : ?x3 = c 3,
    H9 : ?x4 = c 4, H11 : ?x2 = c 5, H13 : ?x3 = c 6,
    H15 : ?x4 = c 7 |- type1_H _ \/ type2_H _ \/ type3_H _
      \/ type4_H _ ] =>
190   type3_H_tac H3 H5 H7 H9 H11 H13 H15

| [ H3 : ?x1 = c 1, H5 : ?x2 = c 2, H7 : ?x3 = c 3,
    H9 : ?x4 = c 4, H11 : ?x2 = c 5, H13 : ?x4 = c 6,
    H15 : ?x3 = c 7 |- type1_H _ \/ type2_H _ \/ type3_H _
      \/ type4_H _ ] =>
195   type4_H_tac_1 H3 H5 H7 H9 H11 H13 H15

| [ H3 : ?x1 = c 1, H5 : ?x2 = c 2, H7 : ?x3 = c 3,
    H9 : ?x2 = c 4, H11 : ?x4 = c 5, H13 : ?x3 = c 6,
    H15 : ?x4 = c 7 |- type1_H _ \/ type2_H _ \/ type3_H _
      \/ type4_H _ ] =>
    type4_H_tac_2 H3 H5 H7 H9 H11 H13 H15
200 | [ H3 : ?x1 = c 1, H5 : ?x2 = c 2, H7 : ?x3 = c 3,
    H9 : ?x4 = c 4, H11 : ?x3 = c 5, H13 : ?x2 = c 6,
    H15 : ?x4 = c 7 |- type1_H _ \/ type2_H _ \/ type3_H _
      \/ type4_H _ ] =>
    type4_H_tac_3 H3 H5 H7 H9 H11 H13 H15

end.
205

Lemma coloring_triple:
  forall c: Coloring, is_good_coloring c H ->
    type1_H c \/ type2_H c \/ type3_H c \/ type4_H c.
Proof.
210 intros. unfold is_good_coloring in H. unfold is_coloring in H.
    destruct H.
    color_next H 1;
    color_next H 2; try find_contr H0 c;
    color_next H 3; try find_contr H0 c;
    color_next H 4; try find_contr H0 c;
215   color_next H 5; try find_contr H0 c;
    color_next H 6; try find_contr H0 c;
    color_next H 7; try find_contr H0 c;
    find_type H3 H5 H7 H9 H11 H13 H15 c.

```

220 | Qed.
| Close Scope positive.

Список литературы

1. A. de Grey, The chromatic number of the plane is at least 5, [arXiv:1804.02385](#), — 2018.
2. Andrew W. Appel, Software Foundations, Volume 3: Verified Functional Algorithms, <https://softwarefoundations.cis.upenn.edu/vfa-current/>, — 2017.
3. H. Hadwiger, Ueberdeckung des Euklidischen Raumes durch kongruente Mengen, *Portugaliae mathematica*, 4(4), 238-242 (1945).
4. A. Soifer, *The Mathematical Coloring Book*, Springer, 2008, ISBN-13: 9780387746401.
5. Marijn J.H. Heule, Computing Small Unit-Distance Graphs with Chromatic Number 5, [arXiv:1805.12181](#) , — 2018.
6. G. Exoo, D. Ismailescu, The chromatic number of the plane is at least 5 — a new proof, [arXiv:1805.00157](#), — 2018.
7. D. Delahaye, A Tactic Language for the System Coq, *In Proceedings of Logic for Programming and Automated Reasoning*, (LPAR), Reunion Island, volume 1955 of Lecture Notes in Computer Science, 85–95. Springer-Verlag, November 2000