МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«АДЫГЕЙСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Кафедра прикладной математики, информационных технологий и информационной безопасности

«ДОПУСКАЕТСЯ К ЗАЩИТЕ»
Заведующий кафедрой
____________ М.В. Алиев
« __ » __________ 20__ г.

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

Направление подготовки 01.04.02 *Прикладная математика и информатика*
Магистерская программа *«Современная теория игр»*

Тема

Оценки хроматического числа двумерной сферы

Научный руководитель	_______ *(подпись)*	к.т.н., доцент *(уч.степень, уч.звание)*	В.А. Воронов *(ФИО)*	__.__.__ *(дата)*
Руководитель программы магистратуры	_______ *(подпись)*	д.ф.-м.н., профессор *(уч.степень, уч.звание)*	А.В. Савватеев *(ФИО)*	__.__.__ *(дата)*
Обучающийся	2ПМ *(группа)*	факультета математики и компьютерных наук	И.П. Морозов *(ФИО)*	

Майкоп 2020

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Целью настоящей дипломной работы является получение конструктивных верхних оценок для хроматического числа двумерной сферы, то есть построения таких раскрасок сферы, при которых точки, находящиеся на единичном расстоянии, раскрашены по-разному. Это задача примыкает к области классических исследований плотнейших упаковок и редчайших покрытий сфер, находится на стыке теории графов, комбинаторной геометрии и теории кодирования. Актуальность данной работы обусловлена отсутствием конструктивных методов получения раскрасок сфер и слабой изученностью рассматриваемой задачи, в отличие от асимптотики хроматических чисел n-мерных сфер при $n \to \infty$. В работе рассматривается семейство раскрасок, полученных разбиением сферы на области Вороного для решений задачи Томсона.

Для достижения поставленной цели необходимо решить следующие задачи:

1. Свести задачу о раскраске графа в m цветов к задаче булевой выполнимости.
2. Провести исследование алгоритмов и методов, применяемых при построении современных SAT-решателей.
3. Применить SAT-решатели для поиска хроматического числа графов и получить корректные раскраски сфер, вычислить диапазоны радиусов.
4. Получить оценки хроматических чисел сфер на основе решений задачи Томсона о минимуме потенциала системы k точечных зарядов на сфере и разбиений на области Вороного.
5. Доказать нижние оценки для хроматических чисел квадратов двойственных графов (триангуляций Делоне).

Рассматриваемая задача тесно связана с классической задачей Нелсона – Эрдёша – Хадвигера о хроматическом числе $\chi(\mathbb{R}^2)$ континуального графа, вершинами которого являются точки плоскости, причем ребрами соединены вершины, находящиеся на единичном евклидовом расстоянии. Иными словами, требуется раскрасить плоскость в конечное число цветов так, чтобы точки, находящиеся на единичном расстоянии, имели разный цвет. Какое наименьшее число цветов для этого потребуется? В настоящее время известно, что $5 \le \chi(\mathbb{R}^2) \le 7$. Если вторая оценка тривиальна и основана на раскраске гексагонального замощения (Рис. 1), то оценка снизу была получена Хойле [26] лишь в 2018 году при помощи компьютерных вычислений.

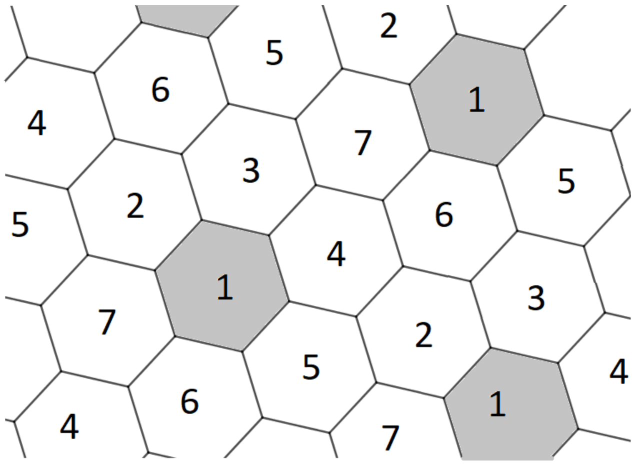

Рис. 1. Раскраска плоскости в 7 цветов.

Аналогичную задачу можно поставить для произвольного метрического пространства. Известно, что $6 \le \chi(\mathbb{R}^3) \le 15$ [32, 33], получены ряд оценок для $\chi(\mathbb{R}^n)$ при $n \ge 3$, а также асимптотика $\chi(\mathbb{R}^n)$ при $n \to \infty$: $(1.239... + o(1))^n \le \chi(\mathbb{R}^n) \le (3 + o(1))^n$ [34, 35]. Также рассматривались хроматические числа рациональных пространств $\mathbb{Q}^n$, многомерных сфер $S^n(r)$, множеств вида $\mathbb{R}^2 \times [0,\ \varepsilon]^k$, ограниченных множеств, точек плоскости с координатами, принадлежащими некоторому квадратичному расширению $\mathbb{Q}$, хроматические числа пространств с запрещенными одноцветными треугольниками. Как правило, полагают, что расстояние между точками множества индуцировано евклидовой метрикой $\mathbb{R}^n$. Основополагающим результатом в данной области является следующая теорема [29].

Теорема (Эрдеш – де Брейне)**.** *Если G – граф с множеством вершин произвольной мощности, и $\chi(G) = k$. Тогда найдется конечный подграф $\widetilde{G} \subseteq G$, для которого $\chi(\widetilde{G}) = k$.*

Это означает, что наилучшую из возможных оценок хроматического числа снизу всегда можно обосновать, перебирая подграфы с конечным числом вершин. Препятствием для осуществления такого перебора (даже в случае $\mathbb{R}^2$) является, во-первых, комбинаторный взрыв, а во-вторых, отсутствие методов, позволяющих раскрасить континуальный граф в тех случаях, когда критический подграф (предположительно) найден.

Отдельный класс задач возникает в том случае, если на раскраску накладываются дополнительные ограничения, например, измеримость каждого из множеств, раскрашенных в один из цветов. Еще более жесткое условие – выпуклость компонент связности одноцветных подмножеств, для плоскости – раскраска многоугольных областей. Тогда для раскраски двумерного гладкого многообразия, удовлетворяющего определенным условиям, требуется не менее 7 цветов [27,28].

ГЛАВА 1. РАСКРАСКИ СФЕР

1.1 Постановка задачи и известные результаты

Графом называется пара $G = (V,\ E)$, где V – вершины графа, $E \subseteq \{\,(u,v)\ :\ u,v \in V\,\}$ – ребра графа. Хроматическим числом графа $\chi(G)$ называется минимальное число k такое, что множество вершин V можно разбить (покрасить) на k непересекающихся классов $V_1 \sqcup V_1 \sqcup \cdots \sqcup V_k = V$ так, что никакое ребро из E не соединяет вершины одного класса. В данной работе рассматривается задача о хроматическом числе двумерной сферы $S^2(r) = \{\,x \in \mathbb{R}^3 : \|x\| = r\,\}$, сформулированная Полом Эрдёшем в 1981 году [39]. Предполагается, что расстояние между точками сферы $x, y \in S^2(r)$ задано евклидовой метрикой в $\mathbb{R}^3$: $d(x,y) = \sqrt{\sum_{i=1}^{3}(x_i - y_i)^2}$. Тогда $\chi(S^2(r)) = \min\{\,k : S^2(r) = V_1 \sqcup \cdots \sqcup V_k,\ x, y \in V_i \Rightarrow \|x - y\| \neq 1\,\}$. Во всех случаях, когда требуются непосредственные вычисления, предполагается, что центр сферы находится в начале координат.

Очевидно, что $\chi(S^2(r))$ зависит от r: если $r < \frac{1}{2}$, то $\chi(S^2(r)) = 1$, в то же время $S^2\left(\frac{1}{2}\right) = 2$ (подходит любая раскраска, в которой диаметрально противоположные точки имеют разный цвет). При $r > \frac{1}{2}$ выполнено $\chi(S^2(r)) > 2$, так как соответствующий континуальный граф $G(S^2(r); 1)$ содержит нечетный цикл, для раскраски которого необходимы по крайней мере 3 цвета. В статье [30] Симмонса был получен следующий результат:

Теорема (Симмонс, 1976)**.**

$$\chi\left(S^2\left(\frac{1}{\sqrt{2}}\right)\right) = 4, \quad \chi(S^2(r)) \geq 4 \ \textit{при}\ r > \frac{1}{\sqrt{3}}.$$

Последнее неравенство было получено вложением в сферу конструкции, аналогичной «свернутому» веретену Мозера (Рис. 2), для раскраски которой необходимо 4 цвета. Отметим, что раскраска сферы $\chi\left(S^2\left(\frac{1}{\sqrt{2}}\right)\right)$ возникает в некой задаче квантовой механики, вследствие чего этот результат был переоткрыт другими авторами. Случай интересен тем, что $d(u,v) = 1$ эквивалентно $(u,v) = 0$. Из неравенства $\chi(\mathbb{R}^3) \leq 15$ следует, что

$$\forall r > 0 \quad \chi(S^2(r)) \leq 15.$$

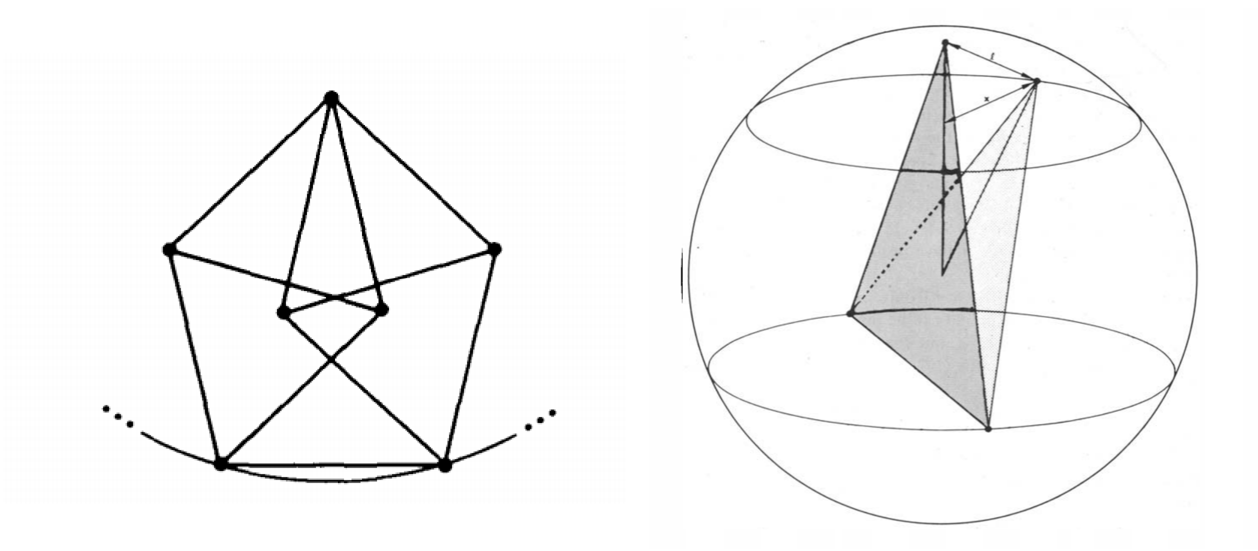

Рис. 2. К теореме Симмонса [30].

В дополнение к предыдущим рассмотрим некоторые асимптотические оценки. В 1983 году Л. Ловас доказал, что $\chi(S^{n-1}(r)) \geq n$. В статье [31] А.М. Райгородский показал, что хроматическое число сферы растет экспоненциально при росте размерности.

Теорема.

$$\textit{Если } \frac{1}{2} < r \leq \frac{1}{\sqrt{2}}, \textit{ то } \chi(S^{n-1}(r)) \geq \left(2\left(\frac{1}{8r^2}\right)^{\frac{1}{8r^2}}\left(1-\frac{1}{8r^2}\right)^{1-\frac{1}{8r^2}} + o(1)\right)^n.$$

$$\textit{Если } r \geq \frac{1}{\sqrt{2}}, \textit{ то } \chi(S^{n-1}(r)) \geq \left(2\left(\frac{1}{4}\right)^{\frac{1}{4}}\left(\frac{3}{4}\right)^{\frac{3}{4}} + o(1)\right)^n.$$

В работе [38] О. Костина доказала усиленный вариант предыдущей теоремы.

Теорема. *Пусть* $r > \frac{1}{2}$ *и* b_1, b_{-1} *таковы, что* $b_1 + b_{-1} \in (0, 1]$ *и* $b_1 < b_{-1}$. *Пусть* $k_1 = [b_1 n]$ *и* $k_{-1} = [b_{-1} n]$. *Положим*

$$p_0(r, b_1, b_{-1}, n) = \frac{(k_1 + k_{-1})n - (k_1 + k_{-1})^2}{2nr^2}$$

Пусть $p(r, b_1, b_{-1}, n)$ *– минимальное простое число, строго большее, чем* p_0. *Если при данных* r, b_1, b_{-1}, n *выполнено* $k_1 + k_{-1} - 2p < -2k_{-1}$, *то*

$$\chi(S^{n-1}(r)) \geq \frac{\binom{n}{k_1}\binom{n-k_1}{k_{-1}}}{\sum\limits_{(m_1,m_2)\,\in\,\mathcal{A}} \binom{n}{m_1}\binom{n-m_1}{m_2}}$$

где $\mathcal{A} = \{(m_1, m_2) : m_1, m_2 \in \mathbb{N} \cup \{0\}, m_1 + m_2 \leq n, m_1 + 2m_2 \leq p - 1\}$.

Очевидно, что $\chi(S^{n-1}(r)) \leq \chi(\mathbb{R}^n) \leq (3 + o(1))^n$, т.к. $S^{n-1}(r)$ лежит в $\mathbb{R}^n$. В общем случае лучших верхних оценок нет. Однако К.А. Роджерс [36] получил более точную оценку в случае $r < \frac{3}{2}$.

Теорема.

$$\textit{Если } r < \frac{3}{2}, \ \textit{то } \chi(S^{n-1}(r)) \leq (2r + o(1))^n.$$

В работе [37] Р. Просановым получена следующая верхняя оценка хроматического числа сферы:

Теорема. $\chi(S_r^{n-1}) \leq (x(r) + o(1))^n$, *где*

$$x(r) = \begin{cases} \sqrt{5 - \frac{2}{r} + 4\sqrt{1 - \frac{5r^2-1}{4r^4}}}, & r > \frac{\sqrt{5}}{2} \\ 2r, & \frac{1}{2} < r \leq \frac{\sqrt{5}}{2} \end{cases}.$$

Так как шестиугольники в раскраске плоскости (Рис. 1) допускают небольшие деформации, кажется, что «почти плоский» участок сферы большого радиуса может быть раскрашен в 7 цветов. Напрашивается заключение, что при достаточно большом r в 7 цветов может быть покрашена вся сфера, но на самом деле, как будет показано далее, аналогичных раскрасок всей сферы не существует. Тем не менее этот подход позволяет построить корректные раскраски сфер для некоторых диапазонов радиусов и получить на их основе верхние оценки для $\chi(S^2(r))$.

1.2 Разбиение на области Вороного

Рассматриваются раскраски сферы $S^2(r)$ в k цветов. Пусть $c(x)$ – цвет точки x, а C_i – множество точек, раскрашенных в i-й цвет, $\bigcup\limits_{i=1}^{k} C_i = S^2(r)$. Требуется выполнить условие:

$$\forall\, x, y \in S^2(r),\ \|x - y\| = 1 \Rightarrow c(x) \neq c(y) \tag{1}$$

В данной работе в качестве таких одноцветных областей C_i предлагается рассматривать выпуклые сферические многоугольники, построенные как области Вороного для некоторого набора точек. Если задан набор точек (сайтов, генераторов) $x_1, x_2, \ldots, x_m \in S^2(r)$, то диаграммой Вороного называется разбиение $S^2(r) = \bigcup\limits_{i=1}^{m} V_i$, где V_i - ячейки Вороного:

$$V_i = \{y \in S^2(r) : \quad \|y - x_j\| \geq \|y - x_i\|, \quad 1 \leq j \leq n,\, j \neq i\}.$$

Поскольку мы будем красить каждую область Вороного в один цвет и не рассматриваем разбиение на области, которые могут быть объединены с соседними, потребуем

$$\operatorname{Diam} V_i < 1 \wedge \operatorname{Diam}(V_i \bigcup V_j) > 1 \tag{2}$$

В зависимости от контекста будем называть диаграммой Вороного как разбиение на ячейки, так и граф из вершин и ребер, составляющих эти ячейки. Такие мозайки названы в честь русского математика Георгия Вороного, который описал общий n-мерный случай в 1908 году, являются фундаментальным инструментом вычислительной геометрии, широко используются в компьютерной графике, $3D$-моделировании, картографии, геофизике, химии, биологии. Для построения разбиений Вороного предложен ряд эффективных ($O(n \log n)$) алгоритмов и их компьютерных реализаций: *CGAL*, *TRIPACK*, *STRIPACK*, *Triangle*, *QHull*, [40], последний из которых является модификацией алгоритма Кларксона – Шора [44] и использовался в данной работе.

Мозайке Вороного однозначно соответствует граф триангуляции Делоне (состоящий из непересекающихся отрезков, соединяющий заданный набор точек так, что при построении плоскости, проходящей через три точки, которые образуют треугольник, все остальные точки лежат не выше этой плоскости), который можно построить за $O(n)$:

Теорема. *Если соединить все сайты, соответствующие смежным ячейкам диаграммы Вороного, получится (двойственный) граф триангуляция Делоне для этого множества точек.*

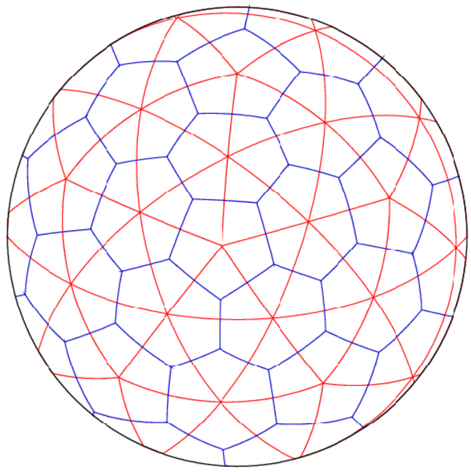

Рис. 3. Диаграмма Вороного и двойственная триангуляция.

На Рис. 3 приведен пример мозайки (синим) и двойственного графа. Отметим, что для заданного набора точек сферическая триангуляция Делоне всегда существует, причём для каждого набора точек, в котором никакие четыре не лежат на одной окружности, она единственна. Пусть задан набор точек и диаграмма Вороного. Обозначим как $\mathcal{G}$ граф триангуляции Делоне, двойственный данной диаграмме.

Назовем *дефектом* вершины $v \in V(G)$ число $\delta(v) = 6 - deg(v)$. Тогда для графа триангуляции сферы $\mathcal{G}$ выполнено

$$\sum_{v_i \in V(\mathcal{G})} \delta(v_i) = 12 \tag{3}$$

Это равенство следует из знаменитой формулы Эйлера для планарных графов: $v - e + f = 2$. В частности, граф $\mathcal{G}$ может содержать 12 вершин степени 5 и сколь угодно большое число вершин степени 6.

Предположим, что каждая из областей Вороного покрашена в некоторый цвет, и эта раскраска удовлетворяет условию (1). Раскрасим вершины $\mathcal{G}$ в цвета соответствующих областей. Тогда справедливы следующие утверждения.

Утверждение 1. *Цвета смежных вершин $\mathcal{G}$ различны.*

Доказательство. Предположим, что это неверно. Тогда объединение двух одноцветных областей имеет диаметр больше 1, а поскольку это множество связно, то оно содержит и две одноцветные точки на единичном расстоянии, что противоречит (1). □

Утверждение 2. *Цвета вершин $\mathcal{G}$, находящихся на расстоянии 2 (в смысле длины кратчайшего пути в графе), различны.*

Доказательство. Поскольку диаметр любой области не превосходит единицы, расстояние между областями, имеющими общего соседа, меньше единицы, значит найдется отрезок длины 1, концы которого лежат в этих областях. Следовательно, они должны иметь разные цвета. □

Пусть $\mathcal{G}^2$ – граф, в котором множество вершин совпадает с множеством вершин $\mathcal{G}$, а к ребрам из $\mathcal{G}$ добавлены ребра, соединяющие вершины, которые в графе $\mathcal{G}$ находились на расстоянии 2:

$$E(\mathcal{G}^2) = \{(u, v) : (u, v) \in E(\mathcal{G}) \vee \exists\, w : (u, w), (w, v) \in E(\mathcal{G})\}.$$

Тогда из построения $\mathcal{G}^2$ и утверждений 1 и 2 вытекает следующее.

Утверждение 3. *Для существования правильной раскраски диаграммы Вороного в k цветов необходимо, чтобы $\chi(\mathcal{G}^2) \leq k$.*

Если расстояние между V_i и V_j, $i \neq j$ меньше единицы тогда и только тогда, когда соответствующие вершины смежны в $\mathcal{G}^2$, то последнее утверждение является необходимым и достаточным условием существования правильной раскраски сферы. Учитывая предыдещие построения и рассуждения, раскраску сферы в k цветов получить из «подходящего» набора точек следующим образом: построить для них диаграмму Вороного, двойственный граф триангуляции $\mathcal{G}$, его квадрат $\mathcal{G}^2$, найти $k = \chi(\mathcal{G}^2)$, проверить выполнение всех необходимых условий и вычислить набор радиусов, при которых они выполняются. Основой для выбора точек-генераторов $\{x_i\}$, соответствующих решениям задачи Томсона, стала следующая гипотеза.

Гипотеза 1. *При некоторых N локальный минимум в задаче Томсона индуцирует разбиение сферы на области Вороного, для которого минимум расстояния между областями, находящимися на расстоянии 2 в двойственном графе, строго больше максимума диаметров областей разбиения.*

Данная гипотеза была подтверждена компьютерными вычислениями. Если d_0 – максимальный из диаметров областей V_i, d_1 – минимальное из расстояний между областями на расстоянии 2, то при выполнении всех остальных ограничений диапазон радиусов можно получить как

$$\left({}^1\!/_{d_1},\ {}^1\!/_{d_0}\right), \text{ при условии } {}^{d_1}\!/_{d_0} < 1 \tag{4}$$

Поиск хроматического числа графа является сам по себе сложной задачей. Для ее решения в данной работе использовались SAT-решатели, которым посвящена вторая глава. Алгоритм поиска $\chi(G)$ вытекает из следующего утверждения.

Утверждение. *Задача раскраски графа $G = (V, E)$ в n цветов эквивалентна задаче булевой выполнимости формулы*

$$F(G,n) = \bigwedge_{v\in V} \left(c_v^1 \vee \cdots \vee c_v^n\right) \wedge \bigwedge_{v\in V,\, i<j\leq n} \left(\bar{c_v^i} \vee \bar{c_v^j}\right) \wedge \bigwedge_{(v,w)\in E,\, i\leq n} \left(\bar{c_v^i} \vee \bar{c_w^i}\right) \tag{5}$$

где переменная c_v^i равна 1 тогда и только тогда, когда вершина v покрашена в цвет i.

1.3 Задача Томсона

Множества точек $\{x_i\}$, «равномерно» распределенных на поверхности сферы, возникают как решения трудоемких задач многомерной оптимизации: задачи Томсона (о минимизации потенциала распределения точечных зарядов на сфере) и задачи о сферических кодах (максимизация минимального из попарных расстояний).

Первая из задач сформулирована Дж. Дж. Томсоном в 1904 году [41] при изучении «пудинговой» модели атома. Необходимо определить минимальную конфигурацию полной потенциальной энергии системы N электронов, ограниченных поверхностью единичной сферы, которые отталкиваются друг от друга силой, определяемой Законом Кулона:

$$V(x_1, \ldots, x_N) = \sum_{i,j} \frac{1}{\|x_i - x_j\|} \to \min_{(x_1,\ldots,x_N)} \tag{6}$$

Спустя век после постановки задача Томсона в трехмерном пространстве решена только для случаев $N = 2, 3, 4, 5, 6, 12$. При $N = 1$ решение тривиально. Два электрона располагаются в диаметрально противоположных точках. Три электрона располагаются на большой окружности сферы в вершинах правильного треугольника. Четыре электрона располагаются в вершинах правильного тетраэдра. Для $N = 5$ в 2010 году было получено строгое компьютерное решение с электронами, находящимися в вершинах треугольной дипирамиды. При $N = 6$ электроны находятся в вершинах правильного октаэдра. При $N = 12$ электроны находятся в вершинах правильного икосаэдра. При других N экстремальность какой-либо конфигурации не доказана, однако для наших целей подходят и субоптимальные решения. Рассмотрим несколько алгоритмов их получения.

Метод решетки [43] позволяет получить расположение точек при $N = 10(m^2 + n^2 + mn) + 2$, где m, n – целые числа, $N < 5000$. Пусть $\zeta = e^{\frac{i\pi}{3}}$, Δ – равносторонний треугольник с вершинами $0, m+\zeta n, \zeta m+\zeta^2 n$. Пересечение решетки Λ с базисом $1, \zeta$ и треугольника Δ – конечное множество точек Δ, которое отображается на грань вписанного в сферу икосаэдра. Остальные грани икосаэдра заполняются поворотом на 180° относительно центра общего ребра соседних треугольников. Далее все полученные вершины радиально проецируются на поверхность сферы, после чего функционал суммарной энергии (6) оптимизируется методом сопряженных градиентов. $\upsilon_1(N)$ – количество целых чисел $l \leq N$, для которых описанный алгоритм позволяет разместить электроны на поверхности сферы (существует решение соответствующего диофантова уровнения) можно грубо оценить как $\upsilon_1(N) = 0.018N + 2.1\sqrt{N} + o(\sqrt{N})$. На Рис. 4 приведен пример расположений зарядов для случаев $N = 132$ (слева) и $N = 1032$, полученных таким способом.

Метод сечения икосаэдра [42] позволяет получить расположения лишь для $\upsilon_2(N) \simeq c \log N$ точек. Пусть ABC – треугольник, и a, b, c – середины его сторон BC, CA, AB. Тогда треугольник abc разбивает ABC на четыре треугольника (2-разрез). Такую операцию можно проделать для всех 20 граней икосаэдра, получив триангуляцию из 80 граней, 120 ребер и 42 вершин, радиальная проекция которой на сферу дает нужный результат.

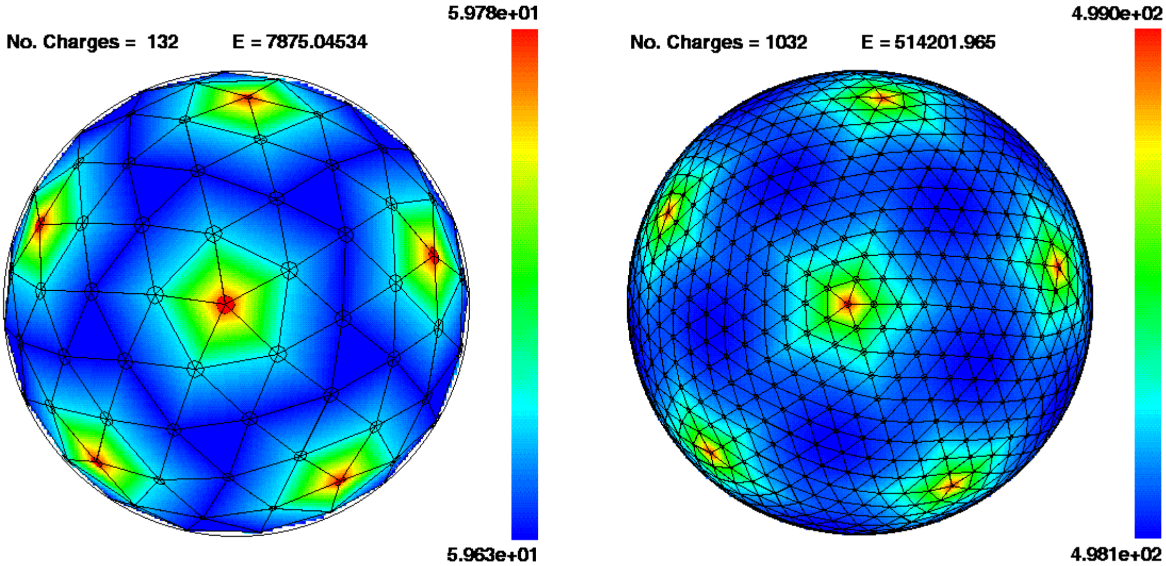

Рис. 4. Субоптимальные решения задачи Томсона [43].

Аналогично строится m-разрез: каждая сторона треугольника делится на m сегментов, порождая m^2 новых треугольников. Эту операцию можно повторять итеративно, получая все более «мелкие» разбиения.

Оба описанных метода позволяют построить «регулярные» семейства конфигураций, обладающие икосаэдральной симметрией (I в нотации Шенфлиса). Такие решения представляют особый теоретический интерес. Их можно параметризовать парой натуральных чисел p, q , где p и q – длины сторон параллелограмма на треугольной сетке, противоположные (острые) углы которого расположены в ближайших друг к другу вершинах степени 5. Для графа при $N = 132$ (Рис. 4) $p = 1, q = 3$. Соответствующий граф будем обозначать $T(p, q)$. Если верна гипотеза Вегнера [51], утверждающая, что для графа G с максимальной степенью вершины Δ выполнено

$$\chi(G^2) \leq \begin{cases} 7, & \Delta \leq 3 \\ \Delta + 5, & 4 \leq \Delta \leq 7 \\ \lfloor {}^{3\Delta}/_{2} \rfloor + 1, & \Delta \geq 8 \end{cases} \tag{7}$$

то $\chi(T^2(p, q)) \leq 11$. Некоторые теоретические оценки хроматических чисел для квадратов графов такого типа обсуждаются в третьей главе.

Для численных экспериментов в данной работе были использованы наилучшие известные на данный момент конфигурации зарядов для различных N ($12 \leq N \leq 400$), опубликованные в открытой коллекции решений различных оптимизационных задач Кембриджского университета *The Cambridge Cluster Database* [45]. Они получены различными способами, которые сводятся к выбору «хорошего» начального расположения и дальнейшей оптимизации: используют градиентные методы, метод Монте-Карло, алгоритм имитации отжига, генетические алгоритмы, их комбинации и модификации [46].

Отметим, что в ходе предварительных исследований рассматривалась идея улучшения имеющихся решений задачи Томсона с тем, чтобы расширить интервалы радиусов (4), которые «покрывают» раскраски, построенные из этих решений. Однако ее реализация связана с оптимизацией сложно устроенных, недифференцируемых функций и осталась за рамками данной работы.

ГЛАВА 2. ЗАДАЧА ВЫПОЛНИМОСТИ БУЛЕВЫХ ФОРМУЛ И SAT-РЕШАТЕЛИ

2.1. Определения

Конъюнктивной нормальной формой (КНФ) называются булева формула вида $\Phi_m(x_1, x_2, \ldots, x_m) = D_1 \wedge D_2 \wedge \cdots \wedge D_k$, где каждый дизъюнкт $D_j = t_{j,1} \vee t_{j,2} \vee \cdots \vee t_{j,n_j}$, и все литералы $t_{j,i}$ – либо переменные, либо их отрицания, причем переменная может встречаться в дизъюнкте не более одного раза. Задача выполнимости КНФ (*ВЫП*, *SAT*) заключается в следующем: для входной формулы Φ_m указанного вида необходимо определить, существует ли набор значений переменных $(\alpha_1, \alpha_2, \ldots, \alpha_m)$ такой, что $\Phi_m(\alpha_1, \alpha_2, \ldots, \alpha_m) = 1$, выполнима ли Φ_m? При этом исследователя часто интересует не только ответ «да» или «нет», но и сам выполняющий набор переменных или доказательство его отсутствия.

Факт принадлежности задачи *ВЫП* к классу $\mathcal{NP}$ является тривиальным. Более содержательное утверждение, знаменитая теорема Кука-Левина, открывшее важность этой задачи для теории сложности вычислений, а впоследствии и для практических приложений, доказал в 1971 году Стивен Кук (тот же результат независимо получило советский математик Леонид Левин). В работе [1] он впервые ввел понятие $\mathcal{NP}$-полной задачи и доказал $\mathcal{NP}$-полноту задачи выполнимости КНФ, что сделало ее первой известной $\mathcal{NP}$-полной задачей. Идея доказательства состоит в построении формулы, которая выполнима тогда и только тогда, когда соответствующая недетерминированная машина Тьюринга, решающая выбранную задачу за полиномиальное время, останавливается с положительным ответом.

Этот результат позволил доказать $\mathcal{NP}$-полноту множества других задач путем полиномиального сведения к ним задачи выполнимости КНФ, что стало стандартным способом доказательства $\mathcal{NP}$-полноты. Так, в своей работе [2] 1972 года Ричард Карп доказал $\mathcal{NP}$-полноту 21 задачи (ныне известных как «список Карпа»), среди которых: задача о клике, задача о выполнимости булевых формул с тремя литералами (*3-SAT*, для которой известен рандомизированный алгоритм ($\mathcal{PPSZ}$) со сложностью $O(1.32216n)$) [3], задача о раскраске графа и другие.

Вокруг исследования задачи выполнимости, ее приложений и обобщений сформировалось международное научное сообщество «*SAT Association*», проводится ежегодная конференция «*International Conference on Theory and Applications of Satisfiability Testing*», публикуется тематический рецензируемый журнал «*Journal on Satisfiability, Boolean Modeling and Computation, JSAT*». Среди перспективных направлений научных исслендований можно отметить изучение задач *CSP* (задача удовлетворения ограничений), *MAX-SAT* (задача поиска максимального количества выполнимых дизъюнктов), *#SAT* и *ALL-SAT* (подсчет количества выполняющих наборов и их поиск), *SMT* (задача выполнимости формул в теориях), *QBF* (задача о булевой формуле с кванторной приставкой), а также конструирование приближенных ($\mathcal{PTAS}$) и рандомизированных алгоритмов.

Пока вопрос о равенстве классов $\mathcal{P}$ и $\mathcal{NP}$ остается открытым, $\mathcal{NP}$-полнота задачи *SAT* свидетельствует о том, что она является вычислительно сложной и у нас нет эффективных детерминированных алгоритмов, позволяющих в общем случае решать ее за разумное время. С другой стороны, ряд важных прикладных задач естественным образом (например, с помощью преобразования Цейтина) сводится к *SAT* и ее обобщениям: задачи из области автоматизации проектирования (*EDA*), логического синтеза, автоматического доказательства теорем, криптографии, биоинформатики, проверки моделей, формальной верификации программ и автоматического конструирования тестовых покрытий, планирования, построения расписаний.

Интерес исследователей, практическая необходимость, развитие вычислительной техники и новые теоретически результаты привели к созданию ряда подходов (и программных пакетов на их основе – *SAT*-решателей), позволяющих во многих случаях эффективно решать задачу выполнимости. Все они так или иначе используют модификации метода направленного перебора и в худшем случае имеют экспоненциальную сложность. Их можно условно разделить на несколько семейств по принципу организации перебора:

1. Алгоритмы, основанные на поиске с возвратом, из которых наиболее удачным оказался *DPLL* [4] и его улучшенный вариант: *CDCL* [5] и

другие подходы, основанные на методе резолюций.

2. Вариации генетических алгоритмов, например, $GASAT$ [6].
3. Подходы, основанные на применении машинного и глубокого обучения, например, *NeuroSAT* [7], который рассматривает задачу выполнимости как задачу классификации и использует для ее решения рекуррентную нейронную сеть.
4. Методы стохастического локального поиска [14], например, *WalkSAT*, базовая идея которого заключается в итеративном выборе по некоторому правилу ложного дизъюнкта и «переворачивании» одного из входящих в него литералов.
5. Алгоритмы символьных вычислений, основанные на преобразовании бинарной диаграммы решений и ее обобщениях [17].
6. Алгоритмы, основанные на построении набора правил вывода и пошаговом преобразовании формулы в эквивыполнимую [17].

Современные SAT-решатели представляют собой многоуровневые комбинации из нескольких подходов, дополненные различного рода эвристиками (например, случайные рестарты), оптимизациями, преобразованиями формулы [19] (например, исключение переменных методом резолюций или с помощью модифицированной процедуры Фурье-Моцкина) с десятками настраиваемых параметров. Объединение различных техник и тонкая настройка параметров позволяют эффективно решать задачи с тысячами переменных и десятками тысяч дизъюнктов. Отдельного внимания заслуживают детали реализации указанных алгоритмов. Так, мемоизация и специализированные структуры данных могут ускорить выполнение некоторых операций в разы. Другие важные аспекты – возможность инкрементального применения, масштабируемость, эффективное распараллеливание, которое может достигаться несколькими способами: портфельный метод (одновременный запуск нескольких экземпляров решателя); метод «разделяй и властвуй», основанный на разбиении формулы на независимые подформулы; параллельный локальный поиск.

Успешность современных алгоритмов при решений индустриальных задач во многом объясняется тем, что они в общем случае имеют регулярную структуру взаимосвязей между переменными. На Рис. 5 приведены визуализации индустриальной и случайно сгенерированной задач, полученные методом Clauset-Newman-Moore [24].

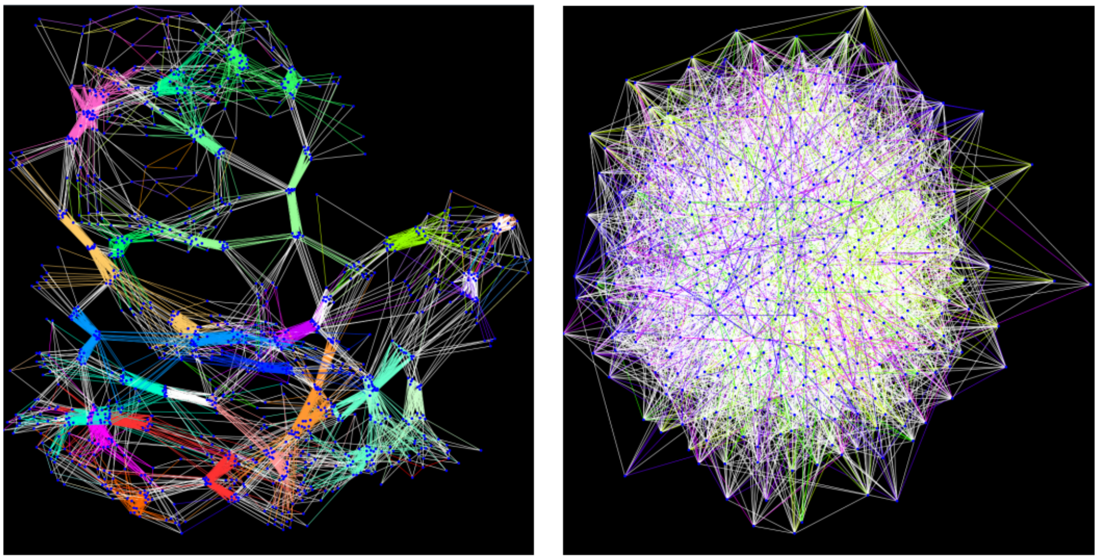

Рис. 5. Сетевая структура индустриальной (слева) и случайно сгенерированной (справа) формул [24].

В качестве примера удачной реализации стратегии «разделяй и властвуй» можно привести проект построения распределенного SAT-решателя *SAT@home* [10], выполненного на платформе для GRID-вычислений *BOINC*, реализованный лабораторией Дискретного анализа и прикладной логики Института динамики систем и теории управления СО РАН, позволивший, в числе прочего, найти несколько ортогональных пар диагональных латинских квадратов порядка 10.

Не все алгоиртмы являются полными в том смысле, что не гарантируют при любом входе завершиться с корректным ответом. Часто отсутствие полноты становится платой за введение дополнительных эвристик, которые могут значительно ускорить решение задач некоторого класса. Многие реализации алгоритма *CDCL* в случае невыполнимости КНФ позволяют построить «сертификат невыполнимости» в одном из общепринятых форматов (*TraceCheck*, *DRUP* [8]), который можно верифицировать специальной утилитой, человеку это сделать обычно не под силу. Так, группе ученых во главе с Марином Хойле удалось с помощью SAT-решателя и суперкомпьютера *Stampede* (800 ядер) построить доказательство в формате *DRAT* отсутствия такой двухцветной раскраски множества $\{1, \ldots, 7825\}$, при которой ни одна пифагорова тройка из этого множества не является одноцветной (булева проблема пифагоровых троек) [9]. Размер файла с доказательством достиг 200 терабайт. Такой метод доказательства утверждений используется все чаще, хотя и не приветствуется математическим сообществом.

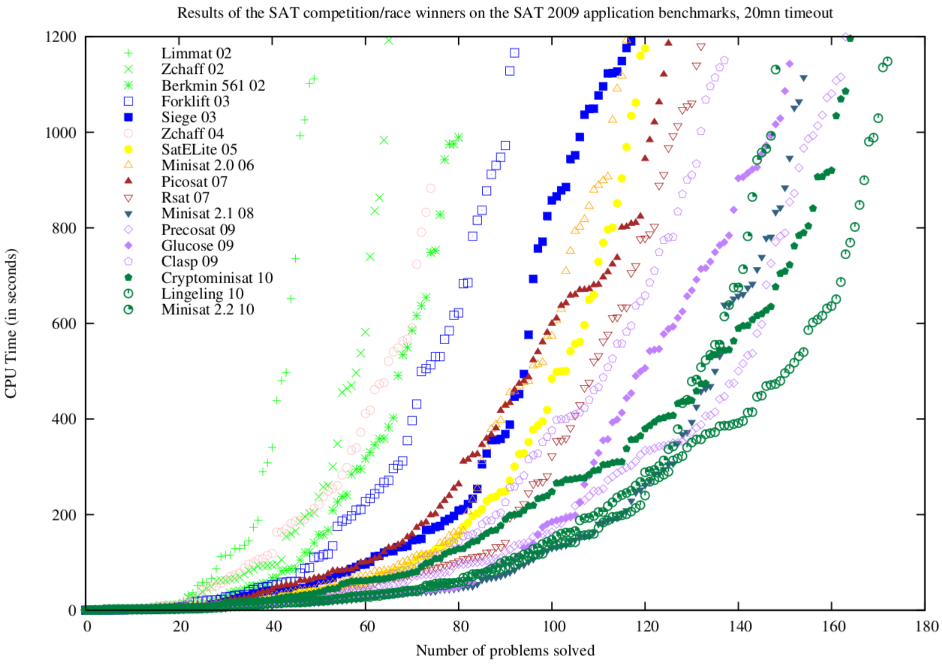

Рис. 6. Результаты The International SAT Competition 2009 года.

На протяжении двух десятков лет проводятся международные состязания по решению задачи *SAT* [11, 12], на которых участники соревнуются в скорости решения специально подобранных задач, записанных в стандартном формате *DIMACS*, при различных условиях (последовательные, параллельные вычисления) и ограничениях. Построение коротких и «сложных» конкурсных задач, а также формализация свойств формулы, которые делают ее сложной для «SAT»-решателя – отдельная интересная проблема. Результаты соревнования (Рис. 6) публикуются на сайте `http://www.satcompetition.org/` (дата обращения: 06.15.2020). Победителями этого соревнования в разные годы становились *MiniSAT*, *Glucose*, *Lingeling*, *CryptoMiniSat*, *YalSAT*, *MapleSAT*, *abcdSAT*, *RISS*. Все эти проекты имеют открытый исходный код и доступны для свободного использования. Отметим, что победителями последних на данный момент соревнований *SAT Race 2019* стал *SAT*-решатель *MapleLCMDiscChronoBT-DL*, предложенный командой ИДСТУ СО РАН (С. Кочемазов, О. Заикин и др.) [47,49]. Группа ученых из Университета Британской Колумбии поддерживает коллекцию *SAT*-задач различного уровня сложности, известную как бенчмарк *SATLIB* [13]. На протяжении многих лет наилучшие результаты на таких соревнованиях, а также и при решении практических задач, показывают алгоритмы, базирующиеся на идее *CDCL*, о который и пойдет речь далее в этой главе.

2.2. Алгоритм DPLL

Алгоритма *DPLL* [4] – это полный и высокоэффективный алгоритм решения задачи выполнимости, основанный на классическом алгоритме решения задач комбинаторной оптимизации: поиск в глубину с возвратом. Он назван в честь своих авторов: Дэвиса, Патнема, Логемана, Лавленда, впервые опубликован в 1962 году и является усовершенствованной версией *DP*, предыдущего алгоритма Дэвиса и Патнема, основанного на методу резолюций.

Далее введем несколько общепринятых в литературе обозначений. Поскольку порядок элементов не важен, здесь и далее формулу Φ удобно представлять как множество дизъюнктов $\{D_1, \ldots, D_k\}$, каждый из которых является множеством литералов $\{t_{j,1}, \ldots, t_{j,n_j}\}$ над переменными из множества X. Если множество дизъюнктов пусто, формула считается тривиально выполнимой, если один из дизъюнктов пуст - не выполнимой. В контексте алгоритмов поиска выполняющих наборов каждая переменная $x \in X$ может находиться в разных состояниях. Переменной может быть присвоено значение $\nu(x)$, $\nu : X \mapsto \{0, 1, ?\}$, где знаком «?» обозначается, что значение переменной не определено. Если $\forall x \in X \ \nu(x) \in \{0, 1\}$, то присваивание называется *полным*, иначе – *частичным*. Присваивание ν позволяет вычислить значения литерала l^ν, дизъюнкта D^ν и всей формулы Φ^ν, в этом случае говорят, что им присвоено соответствующее значение. Переменная называется *чистой*, если она входит в формулу либо только с отрицанием, либо только без отрицания. *Чистую* переменную (и все ее дизъюнкты) можно удалить из формулы, не нарушив ее выполнимость, такая операция называется *удаление чистых переменных*.

В ходе вычислений каждый дизъюнкт, в зависимости от функции присваивания, можно охарактеризовать одним из четырех состояний: *невыполненный*, *выполненный*, *единичный*, *неопределенный*. Дизюнкт называется невыполненным, если всем его литералам присвоен 0, выполненным, если хотя бы одному из его литералов присвоено значение 1, единичным, если всем литералам, кроме одного, значение которого не определено, присвоено значение 1, в остальных случаях дизюнкт считается неопределенным. Конечная цель алгоритма – сделать все дизъюнкты выполненными путем присваивания переменным значений.

Ключевой процедурой алгоритма *DPLL* является *разрешение булевых ограничений*. Если на каком-то этапе вычислений в формуле появился единичный дизъюнкт, то сделать его выполненным можно только одним способом: выбрать подходящее значение неопределеной переменной, которое называется *предполагаемым*. На этом этапе возможен *конфликт*: два разных дизъюнкта могут «потребовать» одновременно противоположных значений переменной. Единичный дизъюнкт ω, который был использован для вывода предполагаемого значения переменной x, называется ее «антецедентом»: $\omega = \alpha(x)$. Если переменная x получила свое значение в результате присваивания, полагают $\alpha(x) = NIL$. В ситуации конфликта присваивания, которые послужили причиной конфликта, отменяются, алгоритм возвращается на шаг назад. Базовый алгоритм является рекурсивным, поэтому можно ввести понятие *уровня присваивания* переменной $\delta(x)$, который равень уровню рекурсии, на котором было выполнено присваивание. Для неопределенных переменных $\delta(x_i) = -1$, для предполагаемых $\delta(x_i) = \max\{\{0\} \cup \{\delta(x_j) : x_j \in \alpha(x_i) \wedge x_j \neq x_i\}\}$. Обозначения $x = v@d$, $d/x = v$ эквивалентны $\delta(x) = d$ и $\nu(x) = v$. На практике разрешение булевых ограничений приводит к каскадному сокращению формулы.

Основная схема алгоритма: по некоторому правилу выбрать из множества неопределенных переменных *переменную ветвления*, присвоить ей некоторое значение, сохранить его в *стеке присваиваний*, преобразовать формулу. Затем рекурсивно проверяется выполнимость новой формулы: если она выполнима, то и исходная формула была выполнимой, алгоритм завершает работу с результатом *SAT*, в противном случае (обнаружен конфликт) запустить ту же процедуру, используя противоположное значение переменной. Если оба значения выбранной переменной привели к конфликту, алгоритм возвращется на шаг назад, выталкивая одно присваивание из стека. Если возвращаться «некуда», алгоритм выдает *UNSAT*. В общем случае алгоритм завершает работу с результатом *UNSAT*, если был выполнен полный перебор всевозможных комбинаций значений переменных.

Преобразование формулы состоит из следующих шагов:

1. Из формулы удаляются все дизъюнкты, которые стали выполненными после присваивания переменной, все остальные вхождения этой переменной удаляются.
2. Выполняется разрешение булевых ограничений.
3. Выполняется удаление чистых переменных.

Псевдокод алгоритма можно записать следующим образом:

```
def DPLL(Φ):
  Φ = preprocess(Φ)
  Φ = pure_literal_elimination(Φ)
  Φ = unit_propagation(Φ)
  if Φ = {}:
    return SAT
  if {} ∈ Φ:
    return UNSAT
  x = choose_literal(Φ)
  return DPLL(Φ ∧ {x}) or DPLL(Φ ∧ {x̄})
```

Доработка этого алгоритма ведется в нескольких направлениях:

1. Построение различных эвристических правил выбора переменной ветвления и соответствующего литерала.
2. Построение ленивых структур данных, позволяющих ускорить отдельные шаги вычисления и сократить объем используемой памяти.
3. Использование «нехронологических» возвратов и «запоминание» конфликтных дизъюнктов.

Последняя идея привела к созданию алгоритма *CDCL*, который является ядром практически всех современных *SAT*-решателей.

2.3. Алгоритм CDCL

Алгоритм *CDCL* [16] (Conflict-Driven Clause Learning – «обучение дизъюнктам, управляемое конфликтами») предложен в 1996 году независимо двумя группами исследователей. К уже известным шагам *DPLL* добавляется построение *имликационного графа*, который позволяет выводить и «запоминать» зависимости между переменными в виде новых дизъюнктов. Такой анализ структуры конфликтов резко сокращает пространство поиска, ускоряет процесс вычисления, дает возможность «не натыкаться» на одни и те же конфликты, а также, в случае невыполнимости, строить более краткие доказательства. В случае конфликта импликационный граф позволяет понять, какие присваивания к нему привели, что открывает возможность «нехронологических» возвратов: возврат к тому уровню рекурсии, который послужил причиной конфликта. Кроме того, этот метод дает некоторые ключи к построению инкрементальных и распределенных *SAT*-решателей: обмен «выученными» дизъюнктами между параллельными процессами оказывается гораздо эффективнее простых портфельных методов.

Импликационный граф представляет собой ориентированный ациклический граф $I = (V_I, E_I)$. Его вершины соответствуют присваиваниям, одна специальная вершина κ обозначает ситуацию конфликта: $V_I \subseteq X \cup \{\kappa\}$. Ребра графа соответствуют андецедентам переменных. Для формального определения ребер введем предикат $\lambda(z,\omega)$ вхождения переменной z в дизъюнкт ω:

$$\lambda(z,\omega) = \begin{cases} 1 & \text{если } z \in \omega \vee \overline{z} \in \omega \\ 0 & \text{иначе} \end{cases}$$

Предикат $\nu_0(z,\omega)$ принимает значение 1 тогда и только тогда, когда z входит в ω и равен 0:

$$\nu_0(z,\omega) = \begin{cases} 1 & \text{если } \lambda(z,\omega) \wedge z \in \omega \wedge \nu(z) = 0 \\ 1 & \text{если } \lambda(z,\omega) \wedge \overline{z} \in \omega \wedge \nu(z) = 1 \\ 0 & \text{иначе} \end{cases}$$

Аналогично,

$$\nu_1(z,\omega) = \begin{cases} 1 & \text{если } \lambda(z,\omega) \wedge z \in \omega \wedge \nu(z) = 1 \\ 1 & \text{если } \lambda(z,\omega) \wedge \overline{z} \in \omega \wedge \nu(z) = 0 \\ 0 & \text{иначе} \end{cases}$$

Наконец, $(z_1, z_2) \in E_I$, если следующий предикат равен 1:

$$\epsilon(z_1,z_2) = \begin{cases} 1 & \text{если } z_2 = \kappa \wedge \lambda(z_1, \alpha(\kappa)) \\ 1 & \text{если } z_2 \neq \kappa \wedge \alpha(z_2) = \omega \wedge \nu_0(z_1,\omega) \wedge \nu_1(z_2,\omega) \\ 0 & \text{иначе} \end{cases}$$

Таким образом, множество ребер графа импликации I можно определить как $V_I = \{(z_1, z_2) : \epsilon(z_1, z_2) = 1\}$. Наконец, каждое ребро $(z_1, z_2) \in E_I$ помечается как $\iota(z_1, z_2) = \alpha(z_2)$.

На Рис. 7 приведен пример импликационного графа для формулы

$$\varphi = \omega_1 \wedge \omega_2 \wedge \omega_3 = (x_1 \vee x_{31} \vee \overline{x_2}) \wedge (x_1 \vee \overline{x_3}) \wedge (x_2 \vee x_3 \vee x_4)$$

и присваиваний $x_{31} = 0@3$, $x_1 = 0@5$.

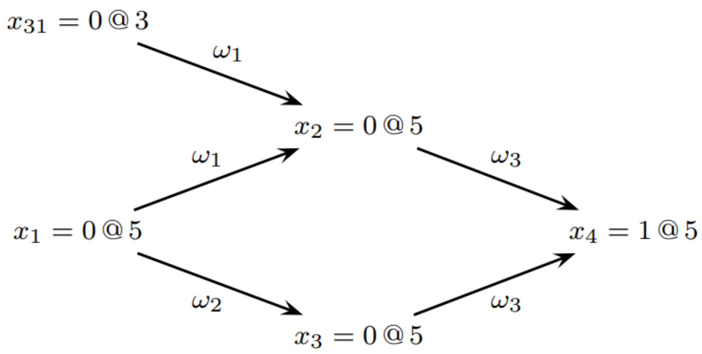

Рис. 7. Пример импликационного графа.

Псевдокод алгоритма *CDCL* можно записать следующим образом:

```
def CDCL(φ, ν):
  if UnitPropagation(φ, ν) == CONFLICT:
    return UNSAT
  dl = 0 # decision level
  while not AllVariablesAssigned(φ, ν):
```

```
        (x, v) = PickBranchingVariable(φ, ν)
        dl = dl + 1
        ν = ν ∪ {(x,v)}
        if UnitPropagation(φ, ν) == CONFLICT:
          β = ConflictAnalysis(φ, ν)
          if β < 0:
            return UNSAT
          else:
            Backtrack(φ, ν, β)
            dl = β
    return SAT
```

К уже известным шагам алгоритма *DPLL* добавлена операция *анализа конфликта*. Эта процедура вычисляет уровень возврата и новую дизъюнкцию. Анализ конфликта состоит в последовательном проходе по графу импликации от вершины κ к антецедентам в сторону уменьшения уровня присваивания и применении правила резолюции (обозначим как $\odot$), что на каждом шаге порождает новую дизъюнкцию. Более формально, определим предикат $\xi(\omega, l, d)$, который равен 1, если в дизъюнкт ω входит предполагаемый лиретал l, который получил свое значение на уровне d:

$$\xi(\omega, l, d) = \begin{cases} 1 & l \in \omega \land \delta(l) = d \land \alpha(l) \neq NIL \\ 0 & \text{иначе} \end{cases}$$

Тогда промежуточные дизъюнкты $\omega_L^{d,i}$, полученные после применения $i = 0, 1, 2, ...$ резолюций можно определеить следующим образом:

$$\omega_L^{d,i} = \begin{cases} \alpha(\kappa) & \text{если } i = 0 \\ \omega_L^{d,i-1} \odot \alpha(l) & \text{если } i \neq 0 \land \xi(\omega_L^{d,i-1}, l, d) = 1 \\ \omega_L^{d,i-1} & \text{если } i \neq 0 \land \forall l \, \xi(\omega_L^{d,i-1}, l, d) = 0 \end{cases}$$

На шаге i, при котором $\omega_L^{d,i} = \omega_L^{d,i-1}$, процесс заканчивается и $\omega_L \triangleq \omega_L^{d,i}$ представляет новый «выученный» дизъюнкт, который добавляется к уже известным. Отметим, что этот процесс очень близко связан с поиском разреза графа импликации. В современных *SAT*-решателях используются ряд дополнительных приемов (*Learned clauses minimization*) для сокращения выученного дизъюнкта [16], и, как правило, используют немного модифицированный подход *1-UIP*, который сводится к поиску первого доминатора импликационного графа, выполняется за линейное время и гарантирует некоторые полезные свойства для выученной дизъюнкции.

2.4. Детали реализации современных SAT-решателей

В этом разделе рассатриваются некоторые технические аспекты реализации SAT-решателей, такие как экристики ветвления, случайные рестарты, наблюдаемые литералы, структура данных, методы подбора параметров, которые сыграли ключевую роль [22] в успешности современных алгоритмов. Стоит отметить, что это далеко не исчерпывающий список приемов и их всестороннему исследованию посвящено множество работ.

Эвристики ветвления

Эвристикой ветвления называется алгоритм выбора переменной ветвления. Простейший способ, в данном случае, - случайный выбор. Еще один возможный вариант - выбирать ту переменную, присваивание которой порождает как можно больше единичных дизъюнктов. Парадоксально, но случайный выбор нередко позволяет получить результат быстрее прочих, поэтому все эвристики так или иначе включают элемент случайности. С другой стороны, в ходе вычислений решатель накапливает определенную информацию, которую можно использовать при выборе переменной ветвления для того, чтобы ускорить вычислительный процесс, сделать его более направленным и контролируемым, а не полагаться на волю случая.

В литературе предложен ряд эффективных методов, которые используют динамическую информацию о ходе вычислений, структуре конфликтов [18], имеют некоторые теоретические обоснования и широко используются на практике. Ключевая идея: каким-либо образом оценить «важность» переменной, используя имеющиеся данные. Стоит отметить, что вычисление такой характеристики может быть достаточно «дорогим», поэтому нередко умозрительно удачные, но «тяжелые» подходы проигрывают на практике. Далее рассматривается несколько наиболее популярных методов.

На случайно сгенерированных задачах лучшие результаты показывает эвристика, предложенная **Bohm**: на каждом шаге из множества неопредленных переменных выбирается переменная с максимальным значением вектора $H_i(x_i) = (H_1(x_i), \ldots, H_m(x_i))$ (в смысле лексикографического порядка), где $H(x) = \alpha \max\{h_i(x), h_i(\overline{x})\} + \beta \min\{h_i(x), h_i(\overline{x})\}$ и $h_i(x)$ – количество неопределенных дизъюнктов с i литералами, содержащих x. Такая эвристика стремится сделать истинными короткие дизъюнкты (при $x = 1$) либо их уменьшить.

Метод **MOM** (*Maximum Occurrences on Minimum sized clauses*) предлагает выбирать переменную x, которая максимизирует функцию $S(x) = (f^*(x) + f^*(\overline{x})) \cdot 2^k + f^*(x) \cdot f^*(\overline{x})$, где $f^*(l)$ - это количество вхождений литерала l в невыполненные дизъюнкты минимального размера. Этот метод выделяет переменные, которые входят в большое количество коротких дизъюнкций с отрицанием или без (при достаточно большом k) и одновременно.

Эвристика **Jeroslow-Wang** устроена следующим образом. Для литерала l вычисляется $J(l) = \sum_{D \in \Phi} 2^{-|D|}$ Односторонний вариант **JW-OS** предполагает выбор литерала l с наибольшим значением $J(l)$. Двусторонний **JW-TS** – поиск переменной x с наибольшей суммой $J(x) + J(\overline{x})$ и присваивание ей 1, если $J(x) \geq J(\overline{x})$. Такой метод стремится выбирать переменные, которые часто встречаются в коротких дизъюнктах.

Эвристики подсчета литералов устроены значительно проще предыдущих и имеют очевидный интуитивный смысл. Пусть $C_p(x)$ - количество неопределенных дизъюнкций, в которые x входит без отрицания, $C_n(x)$ - с отрицанием. Характеристики $C_p(x)$ и $C_n(x)$ можно учитывать в сумме или по отдельности:

1. Выбрать x, для которого $C_p(x) + C_n(x)$ максимальна (**DLCS**) и присваиванить ей 1, если $C_p(x) \geq C_n(x)$.
2. Выбрать x, для которого $C_p(x)$ $(C_n(x))$ максимальна (**DLIS**) и присвоить ей заналогично предыдущему пункту (**DLIS**) или случайно (**RDLIS**).

Характеристика **VSIDS** (*Variable State Independent Decaying Sum*) основана на анализе конфликтов и вычисляется инкрементально:

1. На старте каждым литералом ассоциируется счетчик с нулевым зна-

чением.

2. При добавлении очередной «выученной» дизъюнкции счетчик, ассоциированный с каждым литералом из этой дизъюнкции, увеличивается на 1.
3. С некоторым периодом все счетчики делятся на константу.

На шаге ветвления выбирается литерал с наибольшим значением счетчика (случайно в случае ничьей). Такая эвристика стремится удовлетворить последние конфликтные дизъюнкты и направляет процесс поиска в сторону их разрешения, что может быть особенно эффективно при решении сложных задач (например, задач с функциональными зависимостями между переменными, в которых конфликты возникают часто). Ее подсчет достаточно прост с вычислительной точки зрения: счетчики обновляются только в случае конфликта.

Метод **LEFV** (*Last Encountered Free Variable*) очень быстр и хорошо подходит для невыполнимых формул: запоминается неопределенная переменная, которую алгоритм «встретил» последней на этапе разрешения булевых ограничений. На этапе ветвления выбирается отмеченная переменная, если ее значение все еще не определено, иначе - случайная.

В *MapleCOMSPS* [50] предложен метод численной оценки **learning rate** - вероятности того, что литерал v породит конфликтный дизъюнкт ($LR(v) = \frac{P(v,I)}{L(I)}$) методом обучения с подкреплением («многорукий бандит»), где I - интервал времени между присваиванием v и возвратом его к неопределенному состоянию, $P(v, I)$ - количество выученных дизъюнктов, в которых v задействован на интервале I, $L(I)$ - общее количество выученных дизъюнктов на интервале I. Максимизация этой оценки ведет к выбору переменной, которая устраняет как можно больше конфликтов.

После выбора переменной ветвления необходимо присвоить ей значение, которое также можно выбирать разными способами. В *MiniSAT* переменной ветвления в первую очередь присваивается 0. В *Rsat* предложен метод «запоминания фазы»: переменной с некоторой вероятностью присваивается значение, которое было получено последним в ходе разрешения булевых ограничений.

Рестарты

Частые случайные рестарты широко используются при решении задач комбинаторной оптимизации, помогают бороться с проблемой «тяжелых» хвостов [16]. Под рестартом подразумевается перезапуск алгоритма в некоторый момент, определяемый политикой рестартов. При этом часть информации, накопленной на предыдущем шаге, можно сохранить: выученные дизъюнкции, статистики, накопленные при выборе переменных ветвлений, другая информация о структуре формулы. Было показано, что при переносе одной выученной дизъюнкции и выполнении некоторых других слабых условий полнота алгоритма сохраняется [21]. Рестарты во многих случаях приводят к построению более кратких доказательств невыполнимости.

Перезапуск алгоритма позволяет ему «не застревать» в локальном подпространстве решений. Элемент случайности при выборе переменной ветвления позволяет продолжить поиск в новом направлении. Эффективность этого метода можно объяснить следующим образом: накопленные в ходе вычислений данные отражают текущее представление решателя о том, в каком порядке необходимо выбирать переменные. При этом в отсутствие рестарта решатель не может полностью их использовать, так как ограничен принятыми ранее решениями.

Принятие решения о рестарте, как правило, основывается на количестве конфликтов, наблюдаемых в текущем запуске: оно ограничивается некоторым растущим рядом. В качестве такого ряда используются [21] арифметические прогрессии (*zChaff*), геометрические прогрессии (*MiniSat*), ряд Луби (*MiniSat*) вида $u * t_i$, где u - числовой параметр и

$$t_i = \begin{cases} 2^{k-1} & i = 2^k - 1 \\ t_{i-2^{k-1}+1} & 2^{k-1} \leq i < 2^k - 1 \end{cases}$$

Удаление дизъюнктов

Неконтролируемое добавление «выученных» дизъюнктов может привести к разрастанию формулы и снизить эффективность всех остальных операций. Поэтому были предложены методы оценки дизъюнктов, который позволяют выбирать в каком-то смысле самые «полезные» из них. Так, в *Glucose* был предложен метод *Literals Blocks Distance (LBD)* [48]:

выученный дизъюнкт разбивается на блоки литералов по уровню присваиваний, характеристика LBD полагается равной количеству блоков. Дизъюнкты, для которых $LBD = 2$ перманентно добавляются к формуле. В *MapleLCMDistChronoBT-DL* [49] перманентно сохраняются многократно встречающиеся конфликтные дизъюнкты. Кроме того, применяются некоторые параметрические пороги, при превышении которых «выученные» дизъюнкты удаляются:

1. Запоминаются дизъюнкты, содержащие не более n литералов.
2. Запоминаются дизъюнкты, содержащие не более n неопределенных литералов.
3. Запоминаются дизъюнкты, содержащие не более n литералов или не более 1 неопределенного литерала.

Наблюдаемые литералы

По некоторым оценкам, эффективные реализации SAT-решателей тратят до 90% времени на процедуру разрешения булевых ограничений. Наблюдение за литералами позволяет драматически ускорить этот процесс. Предположим, что на очередном шаге была выбрана некоторая переменная ветвления и ее значение. Далее необходимо найти все невыполненные дизъюнкты, в которых остался ровно один неопределенный литерал. Последовательное сканирование формулы на каждой итерации не эффективно, поэтому в [20] был предложен метод наблюдения за двумя литералами.

В каждом дизъюнкте отмечается два любых неложных литерала. Очевидно, что до тех пор, пока один из наблюдаемых литералов не стал ложным, данная дизъюнкция не является единичной. Таким образом, к этой дизъюнкции необходимо обращаться только тогда, когда одному из наблюдаемых литералов присвоен 0. Возможны два случая:

1. Дизъюнкт не является единичным, то есть все еще содержит как минимум два неложных литерала, один из которых – наблюдаемый, а другим можно заменить прежний наблюдаемый литерал, поддерживая инвариант: в каждой невыполненной дизъюнкции отмечены два неложных литерала.
2. Дизъюнкт стал единичным. Тогда значение второго наблюдаемого ли-

терала выводится и дизъюнкт помечается как выполненный.

Таким образом сокращается количество обращений к памяти, при этом откат на шаг назад по дереву рекурсии не требует «перевыбора» наблюдаемых литералов. Отмечают, что при корректном управлении памятью этот подход позволяет ускорить и присваивание за счет сокращения количества промахов кэша процессора.

Структуры данных и управление памятью

Базовые структура данных, которыми оперирует SAT-решатель – это множество объектов, представляющих литералы, список дизъюнктов, где каждый элемент представляет собой массив ссылок на свои литералы и словарь, где ключом является переменная, а значением – два списка дизъюнктов, в которые переменная входит с отрицанием и без. Позднее были предложены [23] списки дизъюнктов типа *Head and Tail*, где для каждой дизъюнкции поддерживается два указателя, которые на старте указывают на первый и последний литералы, по ходу вычислений сдвигаются навстречу друг другу, что позволяет ускорить отмену присваивания и поиск единичных дизъюнктов, и $WLCC$, в котором кроме двух ссылок внутри каждой дизъюнкции литералы сортируются по возрастанию уровня присваивания.

Интенсивное добавление и удаление дизъюнкций требует специального обращения с оперативной памятью. Поскольку операции динамического запроса памяти у операционной системы (*malloc* и аналоги) является относительно «дорогими», многие реализации используют свою собственную, и, как правило, ленивую, схему управления памятью, основанную на пулах объектов, списках свободной памяти, кешировании и периодической сборке мусора. При запуске у системы запрашивается блок памяти достаточно большого размера, в котором размещаются объекты. При удалении они помечаются специальным флагом. Периодически производится операция сборки мусора, помеченные объекты физически удаляются из памяти.

Де-факто стандартом при разработке продобных систем стал язык программирования C++ и операционные системы семейства UNIX. C++ позволяет, с одной стороны, описать достаточно высокоуровневые абстракции, с другой – контролировать процессы выделения и освобождения памяти на низком уровне. Так, одна из наиболее идиоматичных реализаций *SAT*-решателя – *MiniSat* – реализована на C++ [15]. Авторы, используя метапрограммирование и шаблоны, имплементировали эффективные примитивы, структуры данных, и удобный программный пользовательский интерфейс (*API*). В данный момент почти каждый новый проект, будь то попытка ввести дополнительную эвристику или принципиально новая конструкция алгоритма, основывается на *MiniSat*. Как правило, предполагается, что *SAT*-решатель будет испрользоваться через интерфейс командной строки, куда также выводится оперативная информация о ходе решения. Стоит также отметить библиотеку-«обертку» *python-sat*, автор которой построил унифицированный программный интерфейс на языке *Python*, предоставляющий примитивы для сохранения, загрузки, генерации, обработки формул и единообразного вызова нескольких *SAT*-решателей.

Выбора стратегии решения

В современных SAT-решателях, как правило, используется широкий набор политик, контролирующих ход решения задачи. Они параметризуются набором числовых и категориальных параметров. Все вместе – набор эвристик и их параметры – называется стратегией решения. В работе [22] была сделана попытка статистически оценить, какие из техник вносят наибольший вклад в «успех» алгоритма путем их последовательного отключения и сравнения результатов. Авторы пришли к выводу, что выбор стратегии решения зависит от конкретной задачи и в общем случае не удается дать каких-либо надежных рекомендаций. Почти все реализации поставляются с встроенным набором стратегий, подобранным авторами, например, в *CryptoMiniSat* их около 20. Если встроенная стратегия оказалась неудачной, исследователь может подобрать параметры вручную, что оказывается отдельной непростой задачей: в эффективном *SAT*-решателе *Kissat* более 100 настраиваемых параметров.

В *SATzilla* [25] предложен неожиданный подход решения проблемы выбора стратегии решения, который основывается на предположении, что для формул, кодирующих задачи из одной области, или близких в смысле некоторой метрики, можно использовать сходные стратегии. Точнее, авторы предлагают рассматривать задачу выбора стратегии как задачу классификации, где объекты – это формулы, а метки классов – стратегии, и решать ее методом k-ближайших соседей. При удачном выборе метрик и подходящей обучающей выборке (ее можно построить на основе бенчмарка *SATLIB*, в котором представлены задачи из целого ряда областей), такой метод автоматического выбора параметров выигрывает у жестких предопределенных стратегий, что доказывают результаты *SAT Competition* [12]. Методы машинного обучения успешно применяются и на других других этапах, например, при выборе переменной ветвления.

ГЛАВА 3. ВЕРХНИЕ ОЦЕНКИ ХРОМАТИЧЕСКИХ ЧИСЕЛ СФЕР

В этой главе приводятся численные результаты и некоторые теоретические рассуждения, продолжающие построения из первой главы. Как и раньше, будем обозначать как G двойственный граф триангуляции для некоторой сферической диаграммы Вороного, удовлетворяющей условиям 2, 4, G^2 – его квадрат. Граф триангуляции для диаграмм Вороного с икосаэдральной симметрией обозначается $T(p, q)$.

3.1 Численные эксперименты

В ходе численных экспериментов были загружены субоптимальные решения задачи Томсона, доступные в *Cambridge Cluster Database*. Для них (с помощью *QHull*) были построены диаграммы Вороного, двойственные графы триангуляции и их квадраты. Задачи о раскраске последних в $7, 8, 9, 10$ цветов были перекодированы в задачи булевой выполнимости (в соответствии с формулой 5), которые подавались на вход нескольким SAT-решателям. Из полученных раскрасок квадратов двойственных графов были восстановлены раскраски сферы, вычислены диапазоны радиусов, при которых эти раскраски корректны (выполнены условия 2, 4). Вышеперечисленные шаги были запрограммированы на ЯП Python [Приложение 2,3]. Наряду со стандартными функциями языка активно использовались библиотеки *numpy*, *scipy*, *python-sat*, библиотека для работы с графами *NetworkX*.

Большая часть вычислений выполнялась на сервере с 14-ядерным процессором Intel© Xeon© E5-4660 v3, операционной системой Ubuntu. Основные результаты вычислений приведены на Рис. 8, где жирным выделены диапазоны радиусов, для которых построены корректные раскраски сферы. Более подробные данные о диапазонах радиусов приведены в Приложении 1. Отметим, что в тех случаях, когда не удалось раскрасить квадрат двойственного графа в 8 цветов, препятствием стала нехватка вычислительных ресурсов, отсутствие раскраски не было доказано.

Отдельно рассматривался случай регулярных конфигураций $T(p, q)$: они были сгенерированы методом сечения икосаэдра [Приложение 4]. Даже

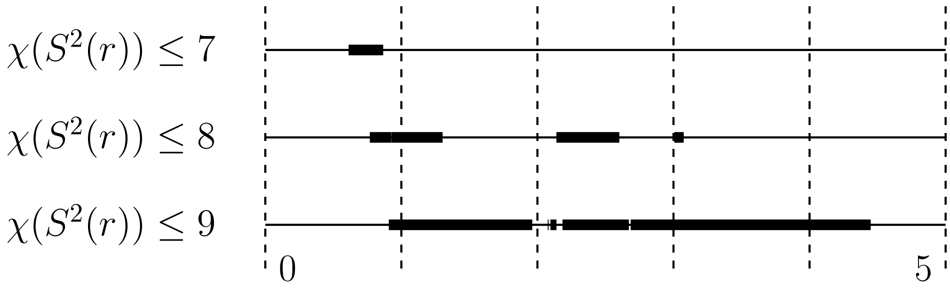

Рис. 8. Диапазоны радиусов.

при малых p, q $(p, q \geq 4)$ эти задачи оказываются достаточно трудоемкими: раскраска в 8 цветов графа $T^2(5, 5)$ потребовала порядка 8 часов работы узла типа «В» вычислительного кластера «Академик В.М. Матросов» ИДСТУ СО РАН, а также применения специализированного SAT-решателя *Cube-and-Conquer*. Автор и научный руководитель благодарят О.С. Заикина за помощь в решении этих задач. Полученные результаты сформулированы в следующем утверждении.

Утверждение 4.

$$\begin{array}{ll} \chi(T^2(2,2)) = 8; & \\ \chi(T^2(2,3)) = 8; & \chi(T^2(3,4)) = 8; \\ \chi(T^2(2,4)) = 8; & \chi(T^2(4,4)) = 8; \\ \chi(T^2(2,5)) = 8; & \chi(T^2(5,5)) = 8. \end{array} \tag{8}$$

Аналогично, раскраски для больших p, q не были построены ввиду ограниченности ресурсов. Этот результат позволяет выдвинуть следующую гипотезу.

Гипотеза 2. *Квадраты двойственных графов регулярных (обладающих икосаэдральной симметрией) (p, q)-решений задачи Томсона являются 8-хроматическими при $p \geq 2, q \geq 2$.*

На разных этапах экспериментов были установлены и протестированы актуальные на момент составления текста версии более 10 SAT-решателей, среди которых *MapleSat*, *MapleCOMSPS-CHB*, *MapleCOMSPS-LRB*, *MapleCOMSPS-pure-CHB*, *MapleCOMSPS-pure-LRB*, *SatElite*, *Cadical*, *CryptoMiniSat5*, *glucose*, *glucose-syrup*, *Kissat*, *MiniSat*, *Picosat*, *lingeling*, *plingeling*, *treen-*

geling, *RSat*, *CnC*. Для компиляции использовался gcc 9.3.0 и опции, указанные авторами. В тех случаях, когда это имело смысл, замерялось время. На Рис. 9 приведены результаты замера времени работы (по горизонтальной оси – количество решенных задач, по вертикальной – суммарное затраченное время) некоторых SAT-решателей (с параметрами по умолчанию) для раскраски в 10 цветов.

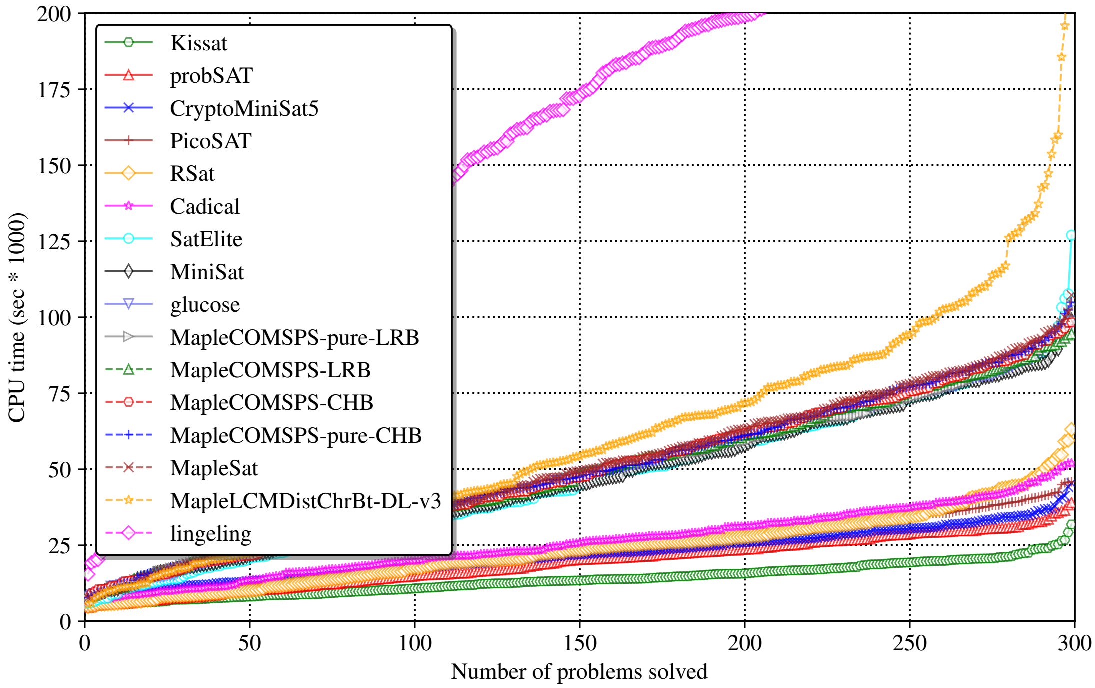

Рис. 9. Сравнение SAT-решателей.

Анализируя эти данные автор пришел к выводу, что сравнение эффективности различных реализаций и комбинаций их параметров на формулах такого вида малоинформативно: многие из них показывают очень близкие результаты. Кое-что можно сказать и о самих задачах. Так, в 10 и 9 цветов почти мгновенно покрасились все примеры, конфликты встречаются редко, при росте N время работы растет незначительно, что говорит о том, что такие задачи оказываются относительно простыми для SAT-решателей. С раскрасками в 8 цветов ситуация прямо противоположная – эта задача оказывается сложной для SAT-решателей и никакие комбинации параметров не позволяют решать ее за разумное время. При этом удаление одной вершины степени 5 и ее соседей снова делает ее простой. Представляется, что проблема возникает в момент «замыкания» раскраски в окрестности одной из таких вершин, решатель сталкивается с неразрешимым конфликтом и возвращается глубоко назад по дереву рекурсии.

Для визуализации построенных графов и раскрасок на языке C++ разработана программа, основанная на графической библиотеке *OpenGL* [Приложение 5]. На Рис. 10 приведены результаты работы программы для раскраски сферы в 9 цветов, $N = 192$.

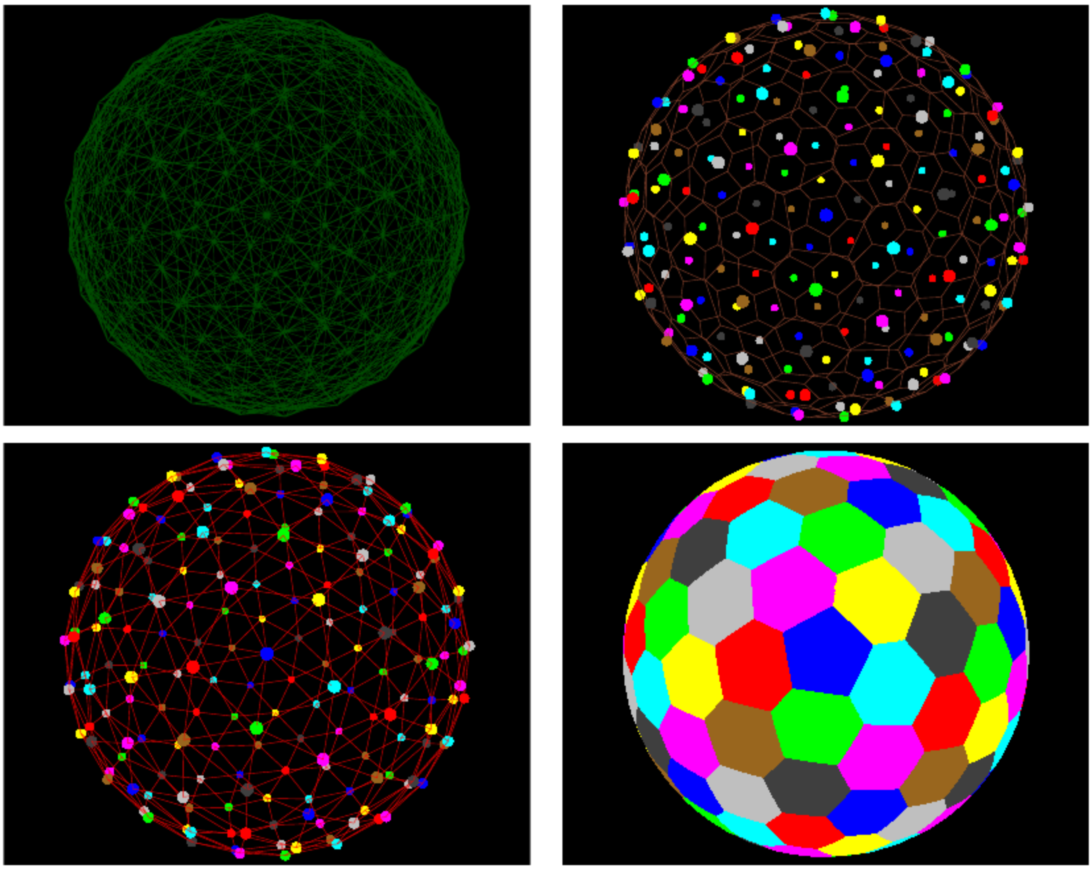

Рис. 10. Пример визуализации раскраски.

3.2 Теоретические оценки

Лемма 1. *Пусть G содержит такую вершину v степени 5, что каждая из смежных с ней вершин имеет степень 6. Тогда $\chi(G^2) \geq 8$. В частности, при $p, q \geq 1$ выполнено $\chi(T^2) \geq 8$.*

Доказательство. На Рис. 11 приведены возможные цвета вершин в окрестности вершины степени 5 (без ограничения общности). Тогда из вершин «внешнего» пояса не более трех имеют цвет 7, а остальные 7 вершин раскрашены в цвета 2–6. Но это невозможно, поскольку из этих 7 вершин можно покрасить в любой из цветов 2–6 не более одной. □

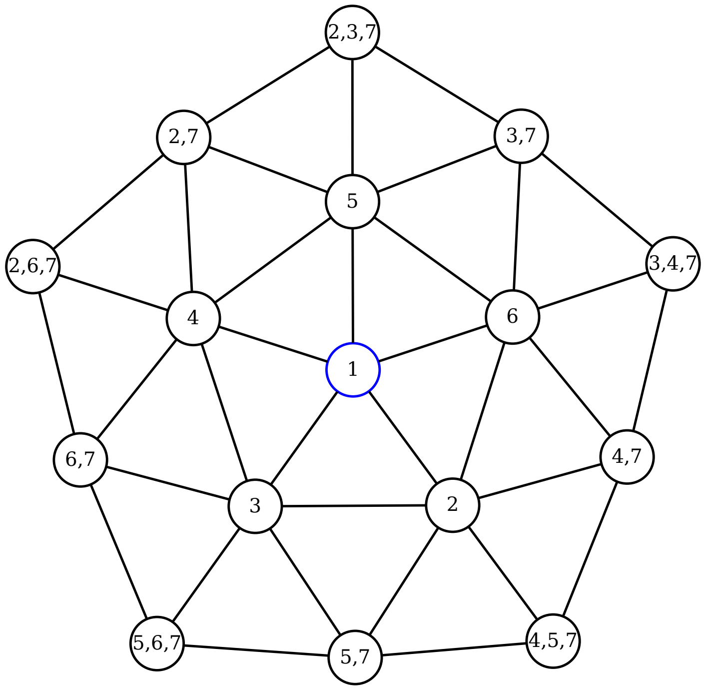

Рис. 11. К лемме 1.

В заключение приведем основной теоретический результат данной работы.

Теорема 1. *При* $r \geq r_0 = \sqrt{\frac{153^2}{1220}}$ *для раскраски областей сферической диаграммы Вороного, удовлетворяющей условиям 2, 4, потребуется по крайней мере 8 цветов.*

Доказательство.

Пусть T – двойственная триангуляция диаграммы Вороного. Предположим, что существует правильная раскраска T^2 в 7 цветов. Тогда T не содержит вершины степени 7 или более (для раскраски ее 1-окрестности потребуется 8 цветов). Поскольку суммарный дефект вершин равен 12, число вершин в ненулевым дефектом не больше 12. Назовем такие вершины *дефектными.*

Заметим, что раскраска полоски ширины 4, состоящей из шестиугольников, определяется однозначно конечным участком (на Рис. 12 последовательно раскрашиваются шестиугольники a, b, c, d). Это рассуждение, разумеется, можно сформулировать и в терминах раскрасок T.

Пусть T_1 – подграф T, полученный как объединение всех 4-окрестностей дефектных вершин; T_2 – подграф T_1, состоящий из вершин T_1, нахо-

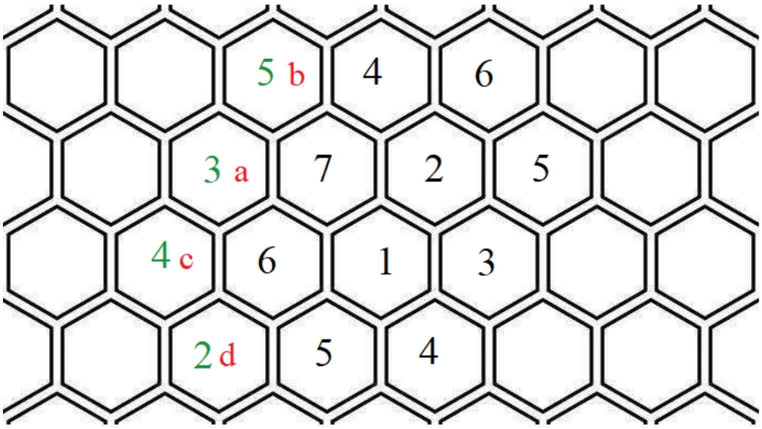

Рис. 12. К теореме 1.

дящихся на расстоянии не более 4 от некоторой вершины из $V(T)\backslash V(T_1)$. Предположим, что T_2 непуст. Очевидно, T_2 является планарным, как и T. Любой цикл на этом графе разбивает графы T, T_2 на две части в смысле укладки на плоскости (сфере) и теоремы Жордана. Обозначим внутреннюю часть $\operatorname{int} L$.

Лемма 2. *Пусть правильная раскраска T в 7 цветов существует, и $L = (v_1, v_2, \ldots, v_l, v_1)$ – некоторый простой цикл на T_2. Тогда сумма дефектов вершин T внутри (снаружи) L делится на 6.*

Доказательство.

Рассмотрим допустимую раскраску T (и вершин L) в 7 цветов. Припишем каждой упорядоченной паре цветов (i, j) направление $\phi_{ij} = k\pi/3$, $k \in \{0, 1, \ldots, 5\}$, соответствующее переходу между данной парой в раскраске плоскости. Каждой паре соседних вершин v_s, v_{s+1} соответствует некоторое направление ϕ_s, причем при обходе L оно остается неизменным, если эта вершина имеет 2-х соседей внутри и 2-х вне L, иными словами, если к ней внутри и вне L примыкает по 3 треугольника триангуляции, порожденной вложением T в плоскость. Если снаружи 4 треугольника, а внутри – 2, то $\phi_{s+1} = \phi_s - \pi/3$ и т. д.:

$$\phi_{s+1} - \phi_s = \frac{\pi}{6} \left(\Delta_{in}(v) - \Delta_{out}(v) \right),$$

где $\Delta_{in}(v)$, $\Delta_{out}(v)$ – число треугольников, примыкающих к вершине, соответственно, внутри и снаружи области, ограниченной путем L.

Назовем дефектом пути число

$$\delta(L) = \sum_s \frac{1}{2}\left(\Delta_{in}(v_s) - \Delta_{out}(v_s)\right).$$

Для любого замкнутого пути на триангуляции из формулы Эйлера следует, что

$$\delta(L) = 6 - \sum_{v \in \operatorname{int} L} \delta(v).$$

Остается заметить, что

$$\sum_{s=1}^{l} \left(\phi(v_{s+1}) - \phi(v_s)\right) = 2\pi k = \frac{\pi}{3}\delta(L),$$

$$\delta(L) = 6k.$$

□

Продолжая доказательство основной теоремы, рассмотрим два случая.

1. Граф T_2 пустой. Тогда каждая 4-окрестность дефектной вершины содержит не более $1+5+10+15+20 = 51$ вершины, а общее число вершин не превосходит $12 \cdot \frac{51}{2} = 306$. Площадь области диаметра 1 на сфере радиуса r оценивается сферху площадью сферического сегмента

$$S_i \leq S_{max} = 2\pi r^2 \left(1 - \sqrt{1 - \frac{1}{4r^2}}\right).$$

Запишем неравенство для площади сферы:

$$612\pi r^2 \left(1 - \sqrt{1 - \frac{1}{4r^2}}\right) \geq 4\pi r^2;$$

$$1 - \sqrt{1 - \frac{1}{4r^2}} \geq \frac{1}{153};$$

$$\sqrt{1 - \frac{1}{4r^2}} \leq \frac{152}{153};$$

$$1 - \frac{1}{4r^2} \leq \frac{152^2}{153^2};$$

$$\frac{1}{4r^2} \geq 1 - \frac{152^2}{153^2} = \frac{305}{153^2};$$

$$4r^2 \leq \frac{153^2}{305};$$

$$r \leq \sqrt{\frac{153^2}{1220}} \approx 4.38.$$

2. Граф T_2 непуст. Тогда число вершин в T, вообще говоря, не ограничено комбинаторными соображениями, но дефектные вершины должны образовывать два подмножества с общим дефектом 6 в каждом, а длина пояса из шестиугольников между этими двумя группами вершин не превосходит 20, и этот пояс покрывает «экватор» сферы. Следовательно,

$$20r \cdot 2 \arcsin \frac{1}{2r} \geq 2\pi r;$$

$$\arcsin \frac{1}{2r} \geq \frac{\pi}{20};$$

$$\frac{1}{2r} \geq \sin \frac{\pi}{20};$$

$$r < \frac{1}{2 \sin \frac{\pi}{20}} \approx \frac{10}{\pi} \approx 3.18.$$

□

ЗАКЛЮЧЕНИЕ

В данной работе рассматривалась задача о построении раскрасок двумерных сфер с запрещенными единичными расстояниями. В ходе работы были полностью решены все поставленные задачи и получены следующие эмпирические и теоретические результаты:

1. Установлено, что семейство раскрасок в 8 и 9 цветов можно получить на основе сферических диаграмм Вороного, соответствующих локальным минимумам задачи Томсона.
2. Дан обзор алгоритмов и методов, применяемых при построении современных SAT-решателей.
3. Разработан программный код, конструирующий корректные раскраски двумерной сферы на основе решения задачи Томсона.
4. Получены раскраски и оценки хроматических чисел сфер некоторых для диапазонов радиусов.
5. Получены оценки хроматических чисел двойственных графов для регулярных решений задачи Томсона.
6. Разработана программа для визуализации сферических диаграмм Вороного и их раскрасок.

Программный код и все материалы, необходимые для воспроизведения полученных результатов, размещены в открытом доступе по адресу `https://github.com/xm-repo/sphere`. Несмотря на то, что данная работа является еще одним небольшим шагом к решению задачи о раскраске сфер, полученные результаты нельзя назвать исчерпывающими. Среди открытых вопросов и направлений для дальнейших исследований можно отметить следующие:

1. Доказательство оценки для хроматического числа квадрата двойственного графа регулярных триангуляций сферы.
2. Доказательство общей оценки хроматического числа сферы для достаточно больших значений радиуса.
3. Постановка и решение аналогичных задач для трехмерных сфер.
4. Изучение случая сферы с радиусом $\frac{1}{2} + \varepsilon$.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Cook S. A. The complexity of theorem-proving procedures //Proceedings of the third annual ACM symposium on Theory of computing. – 1971. – С. 151-158.
2. Karp R. M. Reducibility among combinatorial problems //Complexity of computer computations. – Springer, Boston, MA, 1972. – С. 85-103.
3. Rolf D. Improved bound for the PPSZ/Schöning-algorithm for 3-SAT //Journal on Satisfiability, Boolean Modeling and Computation. – 2006. – Т. 1. – №. 2. – С. 111-122.
4. Robinson J. A. Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. Communications of the ACM, vol. 5 (1962), pp. 394–397 //The Journal of Symbolic Logic. – 1967. – Т. 32. – №. 1. – С. 118-118.
5. Marques-Silva J., Malik S. Propositional SAT solving //Handbook of Model Checking. – Springer, Cham, 2018. – С. 247-275.
6. Lardeux F., Saubion F., Hao J. K. GASAT: a genetic local search algorithm for the satisfiability problem //Evolutionary Computation. – 2006. – Т. 14. – №. 2. – С. 223-253.
7. Selsam D. et al. Learning a SAT solver from single-bit supervision //arXiv preprint arXiv:1802.03685. – 2018.
8. Heule M., Biere A. Proofs for satisfiability problems //All about Proofs, Proofs for all. – 2015. – Т. 55. – №. 1. – С. 1-22.
9. Heule M. J. H., Kullmann O., Marek V. W. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer //International Conference on Theory and Applications of Satisfiability Testing. – Springer, Cham, 2016. – С. 228-245.
10. Zaikin O., Kochemazov S., Semenov A. SAT-based search for systems of diagonal latin squares in volunteer computing project sat@ home //2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO). – IEEE, 2016. – С. 277-281.
11. Järvisalo M. et al. The international SAT solver competitions //Ai Magazine. – 2012. – Т. 33. – №. 1. – С. 89-92.

12. Biere A. Cadical, Lingeling, Plingeling, Treengeling and YalSAT entering the sat competition 2018 //Proc. of SAT Competition. – 2018. – C. 13-14.
13. Hoos H. H., Stützle T. SATLIB: An online resource for research on SAT //Sat. – 2000. – T. 2000. – C. 283-292.
14. Hoos H. H., Stützle T. Local search algorithms for SAT: An empirical evaluation //Journal of Automated Reasoning. – 2000. – T. 24. – №. 4. – C. 421-481.
15. Sorensson N., Een N. Minisat v1. 13-a sat solver with conflict-clause minimization //SAT. – 2005. – T. 2005. – №. 53. – C. 1-2.
16. Biere A., Heule M., van Maaren H. (ed.). Handbook of satisfiability. – IOS press, 2009. – T. 185.
17. Puri R., Gu J. A BDD SAT solver for satisfiability testing: An industrial case study //Annals of Mathematics and Artificial Intelligence. – 1996. – T. 17. – №. 2. – C. 315-337.
18. Marques-Silva J. The impact of branching heuristics in propositional satisfiability algorithms //Portuguese Conference on Artificial Intelligence. – Springer, Berlin, Heidelberg, 1999. – C. 62-74.
19. Eén N., Biere A. Effective preprocessing in SAT through variable and clause elimination //International conference on theory and applications of satisfiability testing. – Springer, Berlin, Heidelberg, 2005. – C. 61-75.
20. Moskewicz M. W. et al. Chaff: Engineering an efficient SAT solver //Proceedings of the 38th annual Design Automation Conference. – 2001. – C. 530-535.
21. Kullmann O. Theory and applications of satisfiability testing //SAT. – 2009. – C. 147.
22. Katebi H., Sakallah K. A., Marques-Silva J. P. Empirical study of the anatomy of modern sat solvers //International Conference on Theory and Applications of Satisfiability Testing. – Springer, Berlin, Heidelberg, 2011. – C. 343-356.
23. Nadel A. Backtrack search algorithms for propositional logic satisfiability: Review and innovations. – Hebrew University of Jerusalem, 2002.
24. Newsham Z. et al. SATGraf: Visualizing the evolution of SAT formula structure in solvers //International Conference on Theory and Applications of Satisfiability Testing. – Springer, Cham, 2015. – C. 62-70.
25. Xu L. et al. SATzilla: portfolio-based algorithm selection for SAT //Journal

of artificial intelligence research. – 2008. – Т. 32. – С. 565-606.

26. Heule M. J. H. Computing small unit-distance graphs with chromatic number 5 //arXiv preprint arXiv:1805.12181. – 2018.
27. Soifer A. The mathematical coloring book: Mathematics of coloring and the colorful life of its creators. – Springer Science & Business Media, 2008.
28. Kronk H. V., Mitchem J. A seven-color theorem on the sphere //Discrete Mathematics. – 1973. – Т. 5. – №. 3. – С. 253-260.
29. Bruijn N. G., Erdos P. A colour problem for infinite graphs and a problem in the theory of relations //Indigationes Mathematicae. – 1951. – Т. 13. – С. 371-373.
30. Simmons G. J. The chromatic number of the sphere //Journal of the Australian Mathematical Society. – 1976. – Т. 21. – №. 4. – С. 473-480.
31. Raigorodskii A. M. On the chromatic numbers of spheres in $\mathbb{R}^n$ //Combinatorica. – 2012. – Т. 32. – №. 1. – С. 111-123.
32. Oren Nechushtan. On the space chromatic number. Discrete mathematics, 256(1):499–507, 2002.
33. David Coulson. A 15-colouring of 3-space omitting distance one. Discrete mathematics, 256(1):83–90, 2002.
34. Андрей М. Райгородский. О хроматическом числе пространства. Успехи математических наук, 55(2):147–148, 2000.
35. David G. Larman and Ambrose C. Rogers. The realization of distances within sets in Euclidean space. Mathematika, 19(01):1–24, 1972.
36. Rogers C. A. Covering a sphere with spheres //Mathematika. – 1963. – Т. 10. – №. 2. – С. 157-164.
37. Prosanov R. Chromatic numbers of spheres //Discrete Mathematics. – 2018. – Т. 341. – №. 11. – С. 3123-3133.
38. Костина О. А., Райгородский А. М. О новых нижних оценках хроматического числа сферы //Труды Московского физико-технического института. – 2015. – Т. 7. – №. 2 (26).
39. Erdos P., Graham R. L. Problem proposed at the 6th Hungarian combinatorial conference //Eger. July. – 1981.
40. Larrea V. G. V. Construction of Delaunay Triangulations on the Sphere: A Parallel Approach. – 2011.

41. Thomson J. J. XXIV. On the structure of the atom: an investigation of the stability and periods of oscillation of a number of corpuscles arranged at equal intervals around the circumference of a circle; with application of the results to the theory of atomic structure //The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science. – 1904. – T. 7. – №. 39. – C. 237-265.
42. Katanforoush A., Shahshahani M. Distributing points on the sphere, I //Experimental Mathematics. – 2003. – T. 12. – №. 2. – C. 199-209.
43. Altschuler E. L. et al. Possible global minimum lattice configurations for Thomson's problem of charges on a sphere //Physical Review Letters. – 1997. – T. 78. – №. 14. – C. 2681.
44. Barber C. B., Dobkin D. P., Huhdanpaa H. Qhull: Quickhull algorithm for computing the convex hull //Astrophysics Source Code Library. – 2013.
45. Wales D. J. et al. The Cambridge cluster database. – 2001.
46. Bondarenko A. N., Karchevskiy M. N., Kozinkin L. A. The Structure of Metastable States in The Thomson Problem //Journal of Physics: Conference Series. – IOP Publishing, 2015. – T. 643. – №. 1. – C. 012103.
47. Heule M. J. H., Järvisalo M., Suda M. Proceedings of SAT Race 2019: Solver and Benchmark Descriptions. – 2019.
48. Audemard G., Simon L. Predicting learnt clauses quality in modern SAT solvers //Twenty-first International Joint Conference on Artificial Intelligence. – 2009.
49. Kochemazov S. et al. MapleLCMDistChronoBT-DL, duplicate learnts heuristic-aided solvers at the SAT Race 2019 //SAT RACE 2019. – C. 24.
50. Liang J. H. et al. Learning rate based branching heuristic for SAT solvers //International Conference on Theory and Applications of Satisfiability Testing. – Springer, Cham, 2016. – C. 123-140.
51. Hartke S. G., Jahanbekam S., Thomas B. The chromatic number of the square of subcubic planar graphs //arXiv preprint arXiv:1604.06504. – 2016.

ПРИЛОЖЕНИЕ 1. ДИАПАЗОНЫ ЗНАЧЕНИЙ РАДИУСА ДЛЯ РЕШЕНИЙ ЗАДАЧИ ТОМСОНА

N	χ	r_{min}	r_{max}
12	7	0.6123724360498286	0.8660254030566606
14	7	0.7012189637303037	0.866624518733132
15	8	0.7793808400826431	0.8656389953148873
16	8	0.7684263761279637	0.9255571736431185
23	8	0.9266065297020001	1.0609578149116226
28	8	1.0193198671994046	1.1904870421859908
29	8	1.1118911931137054	1.1928382470918413
30	8	1.103828172201918	1.2182041997693482
32	8	0.99533136788154	1.3033066587870559
111	8	2.212357881832403	2.3346135438916327
122	8	2.1392950159792288	2.449501775328924
135	8	2.5171094352124648	2.5877451534827087
136	8	2.447587588495632	2.6020398401935063
162	8	2.736805681367747	2.8113962147535827
164	8	2.7502302986365557	2.8450497579658958
184	8	2.84287715104243	3.0157139478058337
187	8	2.861928349097998	3.0560627362196784
192	8	2.7249661450014657	3.102751142557401
195	8	3.0058363930722134	3.0753771547084856
197	8	2.8976233113839487	3.1153470029840915
204	8	2.901518149752298	3.178982675181515
212	8	2.9642824789475735	3.2656300632780146
216	8	3.112953637337046	3.275995598668503
222	8	3.2415374276276974	3.2867469170401122
233	8	3.3082648993572685	3.346269481218835
236	8	3.333360609458009	3.379669049745145
244	8	3.361701422031611	3.4308545982500145
275	8	3.5985687355642995	3.649710877908703
282	8	3.330095988029239	3.7562436984942464

N	χ	r_{min}	r_{max}
295	8	3.7228788034921636	3.786847535818198
311	8	3.7593650380637214	3.8926817082821366
314	8	3.7371303147614583	3.93462175844112
317	8	3.726422432591502	3.9428016618278385
367	8	3.99190579971022	4.249084714640408
372	8	3.841095568367235	4.323232416190199
375	8	4.054259242071652	4.290091782561422
400	8	4.178863159337794	4.449448136318128
20	9	0.9552322210490912	0.9684019184222455
22	9	0.9076990380095793	1.049810069186102
27	9	1.0046137417910734	1.1581231511243772
34	9	1.143590646093195	1.304930527120207
36	9	1.2798797747112989	1.3102016113892045
38	9	1.2570414397707683	1.3801714129853129
39	9	1.1374150070398186	1.3796828971803947
40	9	1.158566580485515	1.4020729430044754
41	9	1.2090537112274389	1.4174223482729418
42	9	1.2662387614856856	1.4345586174481404
45	9	1.447374234588995	1.4852378256091217
46	9	1.346631801514362	1.495266938914718
50	9	1.3896440935146956	1.5960840162398628
51	9	1.573281611556351	1.5772692987432342
56	9	1.595652535666857	1.6608424224821425
58	9	1.6302428305546681	1.6857859722500796
62	9	1.6606746964615353	1.7525995750841878
67	9	1.6828172987995853	1.82256677170873
68	9	1.705482381950591	1.8220306955646062
69	9	1.8204826102695895	1.820678259874453
72	9	1.6620564896383414	1.9191332930682803
75	9	1.8814218812821752	1.9049130498982851
77	9	1.878385308179857	1.9449207721566744
78	9	1.7730755729164913	1.9621470622525556

N	χ	r_{min}	r_{max}
88	9	2.0756380532383494	2.0825696270197906
92	9	2.0954026638173175	2.1399873411539225
107	9	2.221137165656961	2.272252748957184
112	9	2.1840614573093817	2.3558945465936105
113	9	2.187158802364712	2.3595276881801657
115	9	2.2793434035282023	2.3524530209661334
127	9	2.3789716301202626	2.5114023425105523
128	9	2.4933247257758806	2.4990791682828184
132	9	2.2977315371836533	2.5862275269271904
137	9	2.4592235602983035	2.5855965185070424
146	9	2.5697820123477357	2.6719021087062877
160	9	2.6854624025078295	2.775392626525233
161	9	2.7450629229193	2.7717942991901636
177	9	2.7411063865141485	2.949599139846946
182	9	2.827393724885202	2.9995678593229447
188	9	2.9453318480010737	3.016831277347484
200	9	3.115751501406568	3.1285125569081824
202	9	3.00056508912289	3.135728972792108
209	9	3.1166508955872056	3.1906647171577056
214	9	3.047120652177433	3.2627704558154718
217	9	3.095319709660984	3.265621591192845
220	9	3.176622111808861	3.279539766998457
223	9	3.179989790277018	3.2935142969213684
224	9	3.2257135319019765	3.294997114114236
242	9	3.4041744299077563	3.4307298966462403
243	9	3.329201645482653	3.4354532681874144
250	9	3.324399194402528	3.5093548308361684
255	9	3.2840960211932506	3.546997679177647
256	9	3.3059727735358138	3.5529537342201105
257	9	3.2828353509437154	3.573897570495309
258	9	3.360064110241416	3.5665641294387487
259	9	3.356534084563132	3.576180156153748

N	χ	r_{min}	r_{max}
263	9	3.3574674048060174	3.5988119922922652
264	9	3.5303622446074416	3.613529580131228
272	9	3.2479524300443976	3.6920279105437777
273	9	3.471103777455965	3.662315088916946
274	9	3.5098241951640623	3.6583403249876576
276	9	3.4626711347373873	3.6982579099058204
277	9	3.6471563235851163	3.6524196159972626
278	9	3.5869150615879164	3.6731535804441853
279	9	3.5959131199981478	3.6804256458959355
287	9	3.7211497733684866	3.721424249430957
288	9	3.6243345761291996	3.7513569556452264
291	9	3.589989960494795	3.785220806657392
292	9	3.5549133441619825	3.786237548829822
293	9	3.557209395382303	3.808151184595572
299	9	3.7282035049944264	3.8019039222467694
304	9	3.713236032410276	3.8484715564187724
305	9	3.7395550485483353	3.8506719173411037
306	9	3.6586960278980984	3.8911190055267184
308	9	3.686198820686234	3.897642998973312
315	9	3.6916241962952916	3.947070262499115
316	9	3.7774173750757263	3.943345674962319
318	9	3.780896519239086	3.955933141393098
320	9	3.8329705356293515	3.9500458689396263
322	9	3.81217777387446	3.972016673210202
324	9	3.8473829200425427	3.969229913818113
327	9	3.9416975926750353	3.987110181065034
328	9	3.808033857771553	4.007346238741135
329	9	3.8589052406508273	4.010903315769737
330	9	3.8669654421498647	4.029802976479282
331	9	3.8612215197125908	4.009510505471454
332	9	3.8956870683259006	4.0319750383225985
333	9	3.8715004679486476	4.028419743285473

N	χ	r_{min}	r_{max}
339	9	3.94563396667274	4.055849145883436
340	9	4.0324475926749015	4.035034582166366
344	9	3.9758764453660054	4.088398714023686
345	9	4.07364039366915	4.085135336245742
346	9	4.025451688033153	4.082797800082753
348	9	3.8175778625851056	4.15211907479343
349	9	4.078584000628203	4.1024770247652445
356	9	3.907906368339495	4.197006792711948
357	9	3.8602933165289515	4.207005210997042
358	9	3.9076988119905454	4.209311084069152
390	9	4.0487617046216595	4.409909682102864
392	9	3.9511356594065847	4.429186846675495

ПРИЛОЖЕНИЕ 2. КОДИРОВАНИЕ ЗАДАЧИ РАСКРАСКИ ГРАФА

```
import math
import numpy as np
from numpy import linalg as LA
import networkx as nx

#import matplotlib.pylab as plt

#from google.colab import files
#files.upload()
#files.download("file.txt")

#https://pysathq.github.io/
#!pip install python-sat
import pysat
from pysat.formula import CNF
from pysat.solvers import *

import os
import sys
import random
from zipfile import ZipFile
from tqdm import tqdm_notebook as tqdm
from itertools import combinations, permutations

out_dir = ""

class Utils:

    def read_dimacs_graph(file = 'graph.col'):

        if not (os.path.exists(file) and os.path.getsize(file) > 0):
            raise Exception("File " + file + " not found")

        nodes = []
        edges = []
        labels = []
        got_labels = False
        nnodes = nedges = 0

        with open(file, 'r') as f:
            for line in f:
                line = [l.strip() for l in line.split(' ')]
                if line[0] == 'c': #comment
                    continue
                elif line[0] == 'p': #problem
                    nnodes = int(line[2])
                    nedges = int(line[3])
                    nodes = list(range(1, nnodes + 1))
                    labels = [0] * nnodes
                elif line[0] == 'e': #edge
                    edges.append((int(line[1]), int(line[2])))
                elif line[0] == 'l':
                    labels[int(line[1]) - 1] = int(line[2])
                    got_labels = True

        if got_labels:
            nodes = [(n, {'c' : l}) for n, l in zip(nodes, labels)]

        g = nx.Graph()
        g.add_nodes_from(nodes)
        g.add_edges_from(edges)
        return g

    def write_dimacs_graph(file = 'graph.col', g = nx.Graph(), comments =
[]):
        with open(file, 'w') as f:
            for c in comments:
                f.write("c " + c + "\n")
            f.write("p EDGE {} {}\n".format(g.number_of_nodes(), g.
number_of_edges()))
            for u, v in g.edges():
                f.write("e {} {}\n".format(u, v))
            for node in g.nodes():
```

```
                if 'c' in g.node[node]:
                    f.write("l {} {}\n".format(node, g.node[node]['c']))

    def draw_with_colors(g = nx.Graph()):
        color_map = []
        for node in g.nodes():
            if 'c' in g.node[node]:
                color_map.append(g.node[node]['c'] * 10)
        nx.draw(g, pos = nx.spring_layout(g, scale=2), node_color=color_map, with_labels=True, cmap = plt.cm.jet)

    def write_proof(file = "proof.txt", proof = []):
        with open(file, 'w') as f:
            for p in proof:
                f.write("%s\n" % str(p))

    def zip_files(file = "archive.zip", files = []):
        with ZipFile(file, 'w') as archive:
            for f in set(files):
                if not (os.path.exists(file) and os.path.getsize(file) > 0):
                    raise Exception("File " + file + " not found")
                archive.write(f)

def find_triangle(g = nx.Graph()):
    for a in g:
        for b, c in combinations(g[a], 2):
            if b in g[c]:
                return [a, b, c]
    return []

    #return set(frozenset([a, b, c]) for a in g for b, c in combinations(g[a], 2) if b in g[c])


def find_isolates(g = nx.Graph()):
    isolates = []
    for n in g:
        if g.degree(n) == 0:
            isolates.append(n)
    return isolates

class ColMap:

    def __init__(self, g = nx.Graph(), ncolors = 40):

        self.ncolors = ncolors
        self.cmap = dict()
        self.cunmap = dict()

        i = 1
        for node in g.nodes():
            for color in range(1, ncolors + 1):
                self.cmap[(node, color)] = i
                self.cunmap[i] = (node, color)
                i += 1

    def enc(self, node, color):
        return self.cmap[(node, color)]

    def dec(self, node_color):
        return self.cunmap[node_color]

class ColSAT:

    def __init__(self, g = nx.Graph(), ncolors = 10):

        self.ncolors = ncolors
        self.g = g.copy()
        self.cmap = ColMap(g, ncolors)

    def check_coloring(self):
        for n1, n2 in self.g.edges():
            if 'c' not in self.g.node[n1] or 'c' not in self.g.node[n2]:
                return False
            if self.g.node[n1]['c'] == self.g.node[n2]['c']:
                return False
        return True
```

```

    def apply_model(self):

        check = set()
        for var in self.model[self.model > 0]:
            node, color = self.cmap.dec(var)
            self.g.node[node]['c'] = color
            if (node, color) in check:
                raise Exception("Two colors for one node???")
            else:
                check.add((node, color))

        self.colored = self.check_coloring()

        if self.colored != self.solved:
            raise Exception("Something went wrong!")

        return self.colored

    def build_cnf(self):

        self.formula = CNF()
        colors = list(range(1, self.ncolors + 1))

        for n1, n2 in self.g.edges():
            for c in colors:
                self.formula.append([-self.cmap.enc(n1, c), -self.cmap.enc(n2, c)])

        #specials = [28, 194, 242, 355, 387, 397, 468]
        #ii = 1
        #for n in specials:
        #   self.formula.append([self.cmap.enc(n, ii)])
        #  ii += 1


        for n in self.g.nodes():
            #if not n in specials:
            self.formula.append([self.cmap.enc(n, c) for c in colors])
            for c1 in colors:
                for c2 in colors:
                    if c1 < c2:
                        self.formula.append([-self.cmap.enc(n, c1), -self.cmap.enc(n, c2)])

        return self.formula

    def solve_cnf(self, solver = ''):

        triangle = find_triangle(self.g)
        assumptions = []
        if len(triangle) > 0:
            assumptions = [self.cmap.enc(triangle[0], 1), self.cmap.enc(triangle[1], 2), self.cmap.enc(triangle[2], 3)]

        #Glucose3, Glucose4, Lingeling, MapleChrono, MapleCM, Maplesat, Minisat22, MinisatGH
        #with Glucose4(bootstrap_with=self.formula.clauses, with_proof=True) as ms:
        with Lingeling(bootstrap_with=self.formula.clauses) as ms:
            self.solved = ms.solve(assumptions=assumptions)
            if self.solved:
                self.model = np.array(ms.get_model())
                self.apply_model()
            else:
                self.proof = []#ms.get_proof()
                self.colored = False

        return self.solved

if __name__ == "__main__":

    if len(sys.argv) < 4:
        raise "I need in_file out_file ncolors"

    infile = sys.argv[1]
    outfile = sys.argv[2]
    ncolors = int(sys.argv[3])
    g = Utils.read_dimacs_graph(infile)
```

```
    problem = ColSAT(g, ncolors)
    problem.build_cnf().to_file(outfile)
```

ПРИЛОЖЕНИЕ 3. ВЫЧИСЛЕНИЕ ГРАНИЦ ДИАПАЗОНОВ ЗНАЧЕНИЙ РАДИУСА

```
import os
import numpy as np
import itertools
import networkx as nx
import scipy.optimize as opt
from numpy.linalg import norm
from scipy.spatial import SphericalVoronoi

def read_points2(filename):
    points = []
    f = open(filename, 'r')
    k = int(f.readline())
    f.readline()
    for s in f:
        s = ' '.join(s.split())
        l = s.split(' ')
        points.append(np.array([float(l[1]), float(l[2]), float(l[3])], dtype=np.float64))
    f.close()
    return np.array(points)

def read_triang2(filename):
    points = []
    f = open(filename, 'r')
    k = int(f.readline())
    for s in f:
        s = ' '.join(s.split())
        l = s.split(' ')
        points.append(np.array([int(l[0]), int(l[1]), int(l[2])], dtype=np.int32))
    f.close()
    return np.array(points)

def write_dimacs(g, filename):
    g1 = g #nx.convert_node_labels_to_integers(g, first_label=1)
    f = open(filename, 'w')
    f.write('p edges ' + str(g.number_of_nodes()) + ' ' + str(g.number_of_edges()) + '\n')
    for e in g1.edges():
        f.write('e ' + str(e[0]) + ' ' + str(e[1]) + '\n')
    f.close()

def write_faces(faces, filename):
    f = open(filename, 'w')
    for face in faces:
        for p in face:
            f.write(' '.join(map(str, p)) + '\n')
        f.write(' \n')
    f.close()

'''
def build_g2(g):
    paths = dict(nx.all_pairs_shortest_path(g, 2))

    G2 = G.copy()
    keys = list(paths.keys())
    for key in keys:
        for nn in list(paths.get(key).keys()):
            G2.add_edge(key, nn)

    return G2
'''

def connect_dist2(g):

    gg = g.copy()

    to_connect = []
    for i in gg.nodes():
        for j in gg.nodes():
            if nx.shortest_path_length(gg, i, j) == 2:
```

```
                to_connect.append([i,j])
    for e in to_connect:
        gg.add_edge(e[0], e[1])
    return gg

def dist(u, v):
    return norm(u - v)

# angle between radius vectors
def angle(p1 ,p2):
    c = np.dot(p1, p2) / norm(p1) / norm(p2)
    return np.arccos(np.clip(c, -1, 1))

# mid arc between  x, y
def middle(x, y):
    z = (x + y) / 2
    return z / norm(z)

# area diameter
def get_diam(faces):
    diam = 0.0
    for f in faces:
        ij = itertools.combinations(range(len(f)), 2)
        dists = np.array([dist(f[i], f[j]) for i, j in ij])
        diam = np.max([diam, np.max(dists)])
    return diam

def plane_equation(x, y, z):
    a1 = x[1] - x[0]
    b1 = y[1] - y[0]
    c1 = z[1] - z[0]
    a2 = x[2] - x[0]
    b2 = y[2] - y[0]
    c2 = z[2] - z[0]
    a = b1 * c2 - b2 * c1
    b = a2 * c1 - a1 * c2
    c = a1 * b2 - b1 * a2
    d = (- a * x[0] - b * y[0] - c * z[0])
    return np.array([a, b, c, d])

def foot(x,y,z,v):
    p = plane_equation(x, y, z)
    n = np.array([p[0], p[1], p[2]])
    l = norm(n)
    n = n / l #[n[0]/l,n[1]/l,n[2]/l]
    h = p[0]*v[0] + p[1]*v[1] + p[2]*v[2] + p[3]
    n1 = n * h / l #[n[0]*h/l,n[1]*h/l,n[2]*h/l]
    return v - n1 #[v[0]-n1[0],v[1]-n1[1],v[2]-n1[2]]

def circ_dist(x,y,v,eps=1E-8):
    z = np.array([0.0, 0.0, 0.0])
    f = foot(x,y,z,v)

    l = norm(f)
    f = f / l #[f[0]/l,f[1]/l,f[2]/l]
    a1 = angle(x,f)
    a2 = angle(f,y)
    a3 = angle(x,y)
    d = min(dist(x,v), dist(y,v))
    if abs(a1+a2-a3)<eps:
        d = min(d,dist(f,v))
    return d

# distance between areas
def faces_dist(face1, face2):
    f1 = face1
    f2 = face2
    d = 2.0
    for i in range(len(f1)):
        for j in range(len(f2)):
            v = f1[i]
            x = f2[j]
            if j==len(f2)-1:
                y = f2[0]
            else:
                y = f2[j+1]
            d = min(d,circ_dist(x,y,v))
    for i in range(len(f2)):
```

```
        for j in range(len(f1)):
            v = f2[i]
            x = f1[j]
            if j==len(f1)-1:
                y = f1[0]
            else:
                y = f1[j+1]
            d = min(d,circ_dist(x,y,v))
    return d

# center of the circumscribed circle of a spherical triangle, vertex of the Voronoi diagram
def center(x, y, z):
    c = np.vstack([x, y, z])
    x0 = np.mean(c, axis=0)
    x0 = x0 / norm(x0)
    f = lambda v: np.square(np.dot(v, x - y)) + np.square(np.dot(v, x - z)) + np.square(np.dot(v, v) - 1)
    sol = opt.minimize(f, x0, method='nelder-mead')
    return sol.x

def get_dual(g, points):
    faces = []
    for v in g.nodes():
        neigh = g.neighbors(v)

        try:
            n_cyc = nx.cycle_basis(g.subgraph(neigh))[0]   # adjacent vertex cycle
            n_cyc.append(n_cyc[0])
        except IndexError:
            pass
            #print(nx.cycle_basis(g.subgraph(neigh)))

        face = []   # points of the area
        for i in range(len(n_cyc) - 1):
            c = center(points[v - 1], points[n_cyc[i] - 1], points[n_cyc[i+1] - 1])
            face.append(c)

        faces.append(face)
    return faces

def get_dual2(g, points):
    sv = SphericalVoronoi(points)
    sv.sort_vertices_of_regions()
    faces = []
    for region in sv.regions:
        face = sv.vertices[region]
        faces.append(face)
    return np.array(faces)

# minimum of distances between regions at a distance> 3
def faces_d3_dist2(g, faces):
    n = g.number_of_nodes()
    d = 2.0
    for i in g.nodes():
        for j in g.nodes():
            if nx.shortest_path_length(g, i, j) == 3:
                d = np.min([d, faces_dist(faces[i - 1], faces[j - 1])])
    return d

def faces_d3_dist(g, faces):

    d = 2.0
    paths = dict(nx.all_pairs_shortest_path_length(g, 3))

    for i in g.nodes():
        for j in g.nodes():
            if i in paths:
                if j in paths[i]:
                    if paths[i][j] == 3:
                        d = np.min([d, faces_dist(faces[i - 1], faces[j - 1])])

    return d
```

```python
thomsons = [f.strip() for f in open('thomson/list_of_files.txt')]

thomsons.sort(key=lambda x: int(x.replace('.xyz', '')))

for thomson in thomsons:
    try:
        print(thomson)

        n_thomson = int(thomson.replace('.xyz', ''))
        points = read_points2('thomson/' + thomson)

        f = open('tmp', 'w')
        f.write('3 \n')
        f.write(str(len(points)) + '\n')
        for p in points:
            f.write(' '.join(map(str, p)) + '\n')
        f.close()

        cmd = 'C:\\cpp\\qhull-2019.1\\bin\\qconvex.exe i < tmp > ' + 'thomson1/' + str(n_thomson) + '.g'
        os.system(cmd)

        triangles = read_triang2('thomson1/' + str(n_thomson) + '.g')

        G = nx.Graph()

        for t in triangles:
            t = t + 1
            G.add_edge(t[0], t[1])
            G.add_edge(t[1], t[2])
            G.add_edge(t[0], t[2])

        write_dimacs(G, 'g/' + str(n_thomson) + '.g')

        good = True

        for v,d in G.degree:
            if (d != 5) and (d != 6):
                good = False
                break

        if good:
            write_dimacs(connect_dist2(G), 'g2/' + str(n_thomson) + '.g2')

            faces = get_dual2(G, points)
            d0 = get_diam(faces)
            d1 = faces_d3_dist(G, faces)

            #if d1/d0 > 1:
            print(thomson + ' ' + str(d0) + ' ' + str(d1) + ' ' + str(d1/d0))
            write_faces(faces, 'vor/' + str(n_thomson) + '.vor')
    except Exception as e:
        print('err' + str(e))
```

ПРИЛОЖЕНИЕ 4. ПОСТРОЕНИЕ ДИАГРАММЫ ВОРОНОГО С ИКОСАЭДРАЛЬНОЙ СИММЕТРИЕЙ

```python

import networkx as nx
import planarity as pl
import matplotlib.pyplot as plt
import numpy as np
import sys


#a,b = 2,2

edges_dist2 = True

dodec = nx.dodecahedral_graph()

par = [[],[],[]]
npar = 0

n_contr = 0
#nei = np.zeros([20,3],dtype = 'i')
nei = []

node_list = []
node_cl = []

def valid(x,y,a,b):
    return ((x>=0) and (y>=0) and (x<=a+b) and (y<=a+b) and (x+y>=b) and (
    x+y<=a+2*b))


def hexagon_triangular_grid(a,b,z=0):      # generate aq hexagon on the
    triangular grid with sides a-b-a-b-a-b

    g = nx.Graph()
    for i in range(a+b+1):
        for j in range(a+b+1):
            if valid(i,j,a,b):
#                g.add_node((i,j))
                if valid(i+1,j-1,a,b):
                    g.add_edge((i,j,z),(i+1,j-1,z))
                if valid(i+1,j,a,b):
                    g.add_edge((i,j,z),(i+1,j,z))
                if valid(i,j+1,a,b):
                    g.add_edge((i,j,z),(i,j+1,z))
                if edges_dist2:         #  vertices u,v are adjanced if
    dist(u,v)=2 in the initial graph
                    if valid(i+2,j-2,a,b):
                        g.add_edge((i,j,z),(i+2,j-2,z))
                    if valid(i+2,j-1,a,b):
                        g.add_edge((i,j,z),(i+2,j-1,z))
                    if valid(i+2,j,a,b):
                        g.add_edge((i,j,z),(i+2,j,z))
                    if valid(i+1,j+1,a,b):
                        g.add_edge((i,j,z),(i+1,j+1,z))
                    if valid(i,j+2,a,b):
                        g.add_edge((i,j,z),(i,j+2,z))
                    if valid(i-1,j+2,a,b):
                        g.add_edge((i,j,z),(i-1,j+2,z))
    return g

def fill_paral(a,b):   # indices of vertices that belongs to other hexagon
    global npar
    for i in range(b+1):
        for j in range(a+1):
            par[0].append([j+b-i,i])
    for i in range(b+1):
        for j in range(a+1):
            par[1].append([i,a+b-j])
    for i in range(b+1):
        for j in range(a+1):
            par[2].append([a+b-j,b+j-i])
    npar = len(par[0])-1

```

```
def equiv_nodes(u,v):   # check if two vertices in different hexagons are equivalent
#    global n_contr
    e = (u[2],v[2])
    u1 = [u[0],u[1]]
    v1 = [v[0],v[1]]
    if nei[e[0]].count(e[1])==0:
        return False
    i0 = nei[e[0]].index(e[1])
    i1 = nei[e[1]].index(e[0])
    if par[i0].count(u1)==0:
        return False
    if par[i1].count(v1)==0:
        return False
    res = (par[i0].index(u1)+par[i1].index(v1) == npar)
#    if res:
#        print(u,v)
#        n_contr += 1
    return res

def tr_classes(g):    # defining classes of eqiuivalence for networkx.union

    def merge_classes(c):
        for i in range(len(node_list)-1):
            if c.count(node_cl[i])>0:
                node_cl[i] = c[0]

    cln = 0
    for u in g.nodes():
        cur = []
        for k in range(len(node_list)):
            v = node_list[k]
            if equiv_nodes(u,v):
                if cur.count(node_cl[k])==0:
                    cur.append(node_cl[k])

        node_list.append(u)
        if len(cur)==0:
            cln+=1
            node_cl.append(cln)
        else:
            if len(cur)>1:
                merge_classes(cur)
            node_cl.append(cur[0])

def contract_nodes(u,v):  # finally u,v are in the same class
    i0 = node_list.index(u)
    i1 = node_list.index(v)
    return (node_cl[i0]==node_cl[i1])

def write_dimacs(g,filename):
    g1 = nx.convert_node_labels_to_integers(g,first_label=1)
    f = open(filename,'w')
    f.write('p edges '+str(g.number_of_nodes())+' '+str(g.number_of_edges())+'\n')
    for e in g1.edges():
        f.write('e '+str(e[0])+' '+str(e[1])+'\n')
#    print(g1.degree([0,1,2,3,4,5,6,7,8,9,10]))
    f.close()

def write_cnf(g,n,filename):
    g1 = nx.convert_node_labels_to_integers(g,first_label=1)
    f = open(filename,'w')
    f.write('p cnf '+str(g1.number_of_nodes()*n)+' '+str(g1.number_of_nodes()+g1.number_of_edges()*n)+'\n')
    for e in g1.edges():
        for c in range(n):
            f.write(str(-(int(e[0]-1)*n+c+1))+' '+str(-(int(e[1]-1)*n+c+1))+' 0\n')
    for i in range(g1.number_of_nodes()):
        for c in range(n):
            f.write(str(i*n+c+1)+' ')
        f.write('0\n')

    f.close()
```

```
if __name__ == "__main__":
    try:
        a = int(sys.argv[1])
        b = int(sys.argv[2])
    except:
        print('usage: sphere_triang.py a b \n where a,b are positive integers')
        sys.exit()

    tr = nx.Graph()
    for i in range(20):
        h = hexagon_triangular_grid(a,b,i)
        tr = nx.union(tr,h)

#print(tr.number_of_edges())
#tr = hexagon_triangular_grid(2,1)

    L = pl.check_planarity(dodec)[1]
    for i in range(20):
        k = 0
        nei.append([])
        for v in L.neighbors_cw_order(i):
            nei[i].append(v)
            k += 1

    fill_paral(a,b)

    tr_classes(tr)

#
    tr_sphere = nx.quotient_graph(tr,contract_nodes)

    tr_sphere = nx.quotient_graph(tr,contract_nodes)

    fn = 'sph_'+str(a)+'_'+str(b)+'.cnf'

    print('Generating icosahedral S^2 triangulation T({0},{1}) and CNF for 8-coloring...'.format(a,b),'\n')
    print('Number of nodes (variables): {0} ({1}) '.format(tr_sphere.number_of_nodes(),tr_sphere.number_of_nodes()*8),'\n')
    print('Number of edges: ',tr_sphere.number_of_edges(),'\n')
    print('Number of clauses: ',tr_sphere.number_of_edges()*8+tr_sphere.number_of_nodes(),'\n')
    print(fn+'\n')

#print(n_contr)
#print(len(np.unique(node_cl)))

#write_dimacs(tr_sphere,'tr_sphere_'+str(a)+'_'+str(b)+'_d2.txt')
    write_cnf(tr_sphere, 8, fn)

#nx.draw(tr_sphere)
#plt.show()
```

ПРИЛОЖЕНИЕ 5. ВИЗУАЛИЗАЦИЯ РАСКРАСОК

```
#ifndef UTILS_H
#define UTILS_H

#include <set>
#include <map>
#include <cmath>
#include <string>
#include <vector>
#include <fstream>
#include <iostream>

#include <glm/vec3.hpp>
#include <boost/algorithm/string.hpp>

struct Mouse
{
    int x, y, button, state;
};

struct Vec3d
{
    double x, y, z;
};

struct LonLat
{
    double lon, lat;
};

Vec3d Spherical2Cartesian(const LonLat& lola, const double& R = 1)
{
    double x = R * std::cos(lola.lat) * std::cos(lola.lon);
    double y = R * std::cos(lola.lat) * std::sin(lola.lon);
    double z = R * std::sin(lola.lat);
    return Vec3d{ x, y, z };
}

LonLat Cartesian2Spherical(const Vec3d& vec3d, const double& R = 1)
{
    double lat = std::asin(vec3d.z / R);
    double lon = std::atan2(vec3d.y, vec3d.x);
    LonLat res;
    res.lat = lat;
    res.lon = lon;
    return res;
}

Vec3d split_arc(const Vec3d& pp1, const Vec3d& pp2)
{
    auto p1 = Cartesian2Spherical(pp1);
    auto p2 = Cartesian2Spherical(pp2);

    auto& f1 = p1.lat;
    auto& f2 = p2.lat;
    auto& l1 = p1.lon;
    auto& l2 = p2.lon;

    auto Bx = std::cos(f2) * std::cos(l2 - l1);
    auto By = std::cos(f2) * std::sin(l2 - l1);
    auto f3 = std::atan2(std::sin(f1) + std::sin(f2),
        std::sqrt((std::cos(f1) + Bx) * (std::cos(f1) + Bx) + By * By));
    auto l3 = l1 + std::atan2(By, std::cos(f1) + Bx);

    LonLat res;
    res.lat = f3;
    res.lon = l3;

    return Spherical2Cartesian(res);
}

Vec3d split_arc2(const Vec3d& pp1, const Vec3d& pp2, const double frac,
    const double R = 1)
{
    auto p1 = Cartesian2Spherical(pp1);
    auto p2 = Cartesian2Spherical(pp2);
```

```cpp

    auto& f1 = p1.lat;
    auto& f2 = p2.lat;
    auto& l1 = p1.lon;
    auto& l2 = p2.lon;

    auto beta = std::acos(std::sin(f1) * std::sin(f2)
        + std::cos(f1) * std::cos(f2) * std::cos(l1 - l2));

    auto a = std::sin((1 - frac) * beta) / sin(beta);
    auto b = std::sin(frac * beta) / std::sin(beta);
    auto x = a * std::cos(f1) * std::cos(l1) + b * std::cos(f2) * std::cos(l2);
    auto y = a * std::cos(f1) * std::sin(l1) + b * std::cos(f2) * std::sin(l2);
    auto z = a * std::sin(f1) + b * std::sin(f2);
    auto fi = std::atan2(z, std::sqrt(x * x + y * y));
    auto li = std::atan2(y, x);

    LonLat res;
    res.lat = fi;
    res.lon = li;

    return Spherical2Cartesian(res);
}

std::vector<Vec3d> readXYZ(const std::string& fileName)
{
    std::vector<Vec3d> points;
    std::ifstream inFile(fileName);

    for (std::string line; std::getline(inFile, line); )
    {
        std::vector<std::string> parts;
        boost::algorithm::split(parts, line, boost::is_any_of(" "), boost::token_compress_on);

        if (boost::algorithm::iequals("LA", parts[0]))
        {
            Vec3d point = { std::stod(parts[1]), std::stod(parts[2]), std::stod(parts[3]) };
            points.push_back(point);
        }
    }

    inFile.close();

    return points;
}

std::map<int, int> readColoring(const std::string& fileName)
{
    std::map<int, int> coloring;
    std::ifstream inFile(fileName);

    for (std::string line; std::getline(inFile, line); )
    {
        std::vector<std::string> parts;
        boost::algorithm::split(parts, line, boost::is_any_of(" "), boost::token_compress_on);

        if (boost::algorithm::iequals("l", parts[0]))
        {
            coloring[std::stoi(parts[1])] = std::stoi(parts[2]);
        }
    }

    inFile.close();

    return coloring;
}

std::vector<std::pair<int, int>> readTriangEdges(const std::string& fileName)
{
    std::vector<std::pair<int, int>> edges;
    std::ifstream inFile(fileName);

```

```
    for (std::string line; std::getline(inFile, line); )
    {
        std::vector<std::string> parts;
        boost::algorithm::split(parts, line, boost::is_any_of(" "), boost
    ::token_compress_on);

        if (boost::algorithm::iequals("e", parts[0]))
        {
            edges.push_back({ std::stoi(parts[1]), std::stoi(parts[2]) });
        }
    }

    inFile.close();

    return edges;
}

std::vector<std::vector<Vec3d>> readFaces(const std::string& fileName)
{
    std::vector<std::vector<Vec3d>> faces;

    std::ifstream inFile(fileName);

    std::vector<Vec3d> face;
    for (std::string line; std::getline(inFile, line); )
    {
        boost::trim(line);
        if (line.empty())
        {
            if (!face.empty())
            {
                faces.push_back(face);
                face.clear();
            }
            continue;
        }

        std::vector<std::string> parts;
        boost::algorithm::split(parts, line, boost::is_any_of(" "), boost
    ::token_compress_on);
        Vec3d point = { std::stod(parts[0]), std::stod(parts[1]), std::
    stod(parts[2]) };
        face.push_back(point);
    }

    if (!face.empty())
    {
        faces.push_back(face);
        face.clear();
    }

    inFile.close();

    return faces;
}

std::vector<Vec3d> upgrade_face(std::vector<Vec3d> face)
{
    std::vector<Vec3d> newFace;

    auto l = face.size();
    for (size_t i = 0; i < l; i++)
    {
        auto& p1 = face[i];
        auto& p2 = face[(i + 1) % l];

        for (double d = 0; d <= 1; d += 0.005)
        {
            newFace.push_back(split_arc2(p1, p2, d));
        }
    }

    return newFace;
}

#endif
```

```

#include "Utils.hpp"
#include <gl2ps-1.4.0-source/gl2ps.h>
#include <freeglut/include/GL/glut.h>
#include <glm/vec3.hpp>

Mouse mouse = {0, 0};
double cameraDistance = 3.5;
std::pair<double, double> cameraAngleXY(0, 0), cameraTransXY(0, 0);
std::pair<int, int> screenWH;

std::string fileName;
std::string nColors;
std::vector<Vec3d> points;
std::map<int, int> coloring;
std::set<std::pair<int, int>> sedges, sedges2, sedges3;
std::vector<std::pair<int, int>> edges, edges2;
std::vector<std::vector<Vec3d>> faces, faces2;
std::vector<GLfloat> facesTriangles;

std::string vshader =
"#version 330 core"
"layout(location = 0) in vec3 pos;"
"void main() {"
    "gl_Position.xyz = pos;"
"}";

std::vector<Vec3d> palette =
{
    { 0.0, 0.0, 0.0 },
    { 1.0, 0.0, 0.0 },
    { 0.0, 1.0, 0.0 },
    { 1.0, 1.0, 0.0 },
    { 0.0, 0.0, 1.0 },
    { 0.60, 0.40, 0.12 },
    { 1.0, 0.0, 1.0 },
    { 0.75, 0.75, 0.75 },
    { 0.0, 1.0, 1.0 },
    { 0.25, 0.25, 0.25 },
    { 0.98, 0.625, 0.12 },
    { 0.98, 0.04, 0.7 },
    { 0.60, 0.40, 0.70 },
    { 1.0, 1.0, 1.0 },
};

bool drawPoints = false;
bool drawSphere = false;
bool drawG = false;
bool drawG2 = false;
bool drawFaces = true;
bool drawColors = true;
bool drawAxes = false;
bool drawFacesSkeletons = false;


void DisplayCallback();
void DoDraw();

void save(const std::string& fileName, int type = GL2PS_PDF)
{
    FILE *fp;
    GLint buffsize = 0, state = GL2PS_OVERFLOW;
    fp = fopen(fileName.c_str(), "wb");
    printf("Saving ... \n");
    while (state == GL2PS_OVERFLOW) {
        buffsize += 1024 * 1024;
        gl2psBeginPage("test", "gl2psTestSimple", NULL,
            GL2PS_PDF, GL2PS_SIMPLE_SORT,
            GL2PS_DRAW_BACKGROUND | GL2PS_USE_CURRENT_VIEWPORT,
            GL_RGBA, 0, NULL, 100, 100, 100, buffsize, fp, fileName.c_str
    ());
        DisplayCallback();
        state = gl2psEndPage();
    }
    fclose(fp);
    printf("Done \n");
}

```

```
void DisplayCallback()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT |
   GL_STENCIL_BUFFER_BIT);

    /*
    glPushMatrix();
    glTranslatef(-2, -2, 0);
    DoDraw();
    glPopMatrix();

    glPushMatrix();
    glRotatef(180, 0, 1, 0);
    glTranslatef(-2, 2, 0);
    DoDraw();
    glPopMatrix();

    glPushMatrix();
    glTranslatef(2, 2, 0);
    DoDraw();
    glPopMatrix();

    glPushMatrix();
    glTranslatef(2, -2, 0);
    DoDraw();
    glPopMatrix();
    */

    DoDraw();
    glFlush();
    glutSwapBuffers();
}

void DoDraw()
{
    glPushMatrix();

    //camera
    {
        glTranslatef(cameraTransXY.first, cameraTransXY.second, -
   cameraDistance);
        glRotatef(cameraAngleXY.first, 1, 0, 0);
        glRotatef(cameraAngleXY.second, 0, 1, 0);
    }

    if (drawSphere)
    {
        glColor4f(0.3f, 0.3f, 0.3f, 1.f);
        //glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
        glutSolidSphere(0.99f, 100, 100);
    }

    if (drawPoints)
    {
        int i = 0;
        //auto p = points[15];
        for (auto& p : points)
        {
            if (drawColors)
            {
                const auto& clr = palette[coloring[i + 1]];
                glColor4f(clr.x, clr.y, clr.z, 1.f);
            }
            else
            {
                glColor4f(0.0f, 0.0f, 1.0f, 1.f);
            }

            i++;
            glPushMatrix();
            glTranslatef(p.x, p.y, p.z);
            glutSolidSphere(0.03f, 10, 10);
            glPopMatrix();
        }
    }

    if (drawG)
```

```
    {
        glColor4f(1.0f, 0.0f, 0.0f, 1.5f);

        for (auto& e : edges)
        {
            //if (e.first - 1 != 15)
            //{
            //    continue;
            //}
            auto& p1 = points[e.first - 1];
            auto& p2 = points[e.second - 1];
            glBegin(GL_LINES);
            glVertex3f(p1.x, p1.y, p1.z);
            glVertex3f(p2.x, p2.y, p2.z);
            glEnd();
        }
    }

    if (drawG2)
    {
        glColor4f(0.0f, 0.3f, 0.0f, 1.2f);

        for (auto& e : sedges3)
        {
            //if (e.first - 1 != 15)
            //{
            //    continue;
            //}
            auto& p1 = points[e.first - 1];
            auto& p2 = points[e.second - 1];
            glBegin(GL_LINES);
            glVertex3f(p1.x, p1.y, p1.z);
            glVertex3f(p2.x, p2.y, p2.z);
            glEnd();
        }
    }

    if (drawFacesSkeletons)
    {
        glPushMatrix();

        for (auto& face : faces)
        {
            glColor4f(0.6f, 0.3f, 0.2f, 1.f);
            for (size_t i = 0; i < face.size(); i++)
            {
                auto& p1 = face[i];
                auto& p2 = face[(i + 1) % face.size()];

                glBegin(GL_LINES);
                glVertex3f(p1.x, p1.y, p1.z);
                glVertex3f(p2.x, p2.y, p2.z);
                glEnd();
            }
        }

        glPopMatrix();
    }

    if (drawFaces)
    {
        glPushMatrix();

        glLineWidth(2.f);

        int i = 0;
        glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
        //auto& face = faces[15];
        for (auto& face : faces2)
        {
            auto l = face.size();

            if (drawColors)
            {
                const auto& clr = palette[coloring[i + 1]];
                glColor4f(clr.x, clr.y, clr.z, 1.f);
            }
```

```
            else
            {
                glColor4f(0.6f, 0.3f, 0.2f, 1.f);
            }

            auto& c = points[i];
            std::vector<GLfloat> triangles;
            for (size_t j = 0; j < l; j++)
            {
                auto& p1 = face[j];
                auto& p2 = face[(j + 1) % l];

                //glBegin(GL_TRIANGLES);
                //glVertex3f(p1.x, p1.y, p1.z);
                //glVertex3f(p2.x, p2.y, p2.z);
                //glVertex3f(c.x, c.y, c.z);
                //glEnd();

                triangles.push_back(p1.x);
                triangles.push_back(p1.y);
                triangles.push_back(p1.z);

                triangles.push_back(p2.x);
                triangles.push_back(p2.y);
                triangles.push_back(p2.z);

                triangles.push_back(c.x);
                triangles.push_back(c.y);
                triangles.push_back(c.z);
            }
            glEnableClientState(GL_VERTEX_ARRAY);
            glVertexPointer(3, GL_FLOAT, 0, triangles.data());
            glDrawArrays(GL_TRIANGLES, 0, triangles.size() / 3);
            glDisableClientState(GL_VERTEX_ARRAY);


            /*
            glBegin(GL_POLYGON);
            for (auto& p : face)
            {
                glVertex3f(p.x, p.y, p.z);
            }
            glEnd();
            */

            i++;
        }
        glPopMatrix();
    }

    if (drawAxes)
    {
        glPushMatrix();

        glutSolidSphere(.01f, 100, 100);

        //x
        glColor4f(0.0f, 0.3f, 0.0f, 1.5f);
        glBegin(GL_LINES);
        glVertex3f(-2, 0, 0);
        glVertex3f(2, 0, 0);

        //arrow
        glVertex3f(2.0, 0.0f, 0.0f);
        glVertex3f(1.8, 0.1f, 0.0f);
        glVertex3f(2.0, 0.0f, 0.0f);
        glVertex3f(1.8, -0.1f, 0.0f);

        glEnd();

        //y
        glColor4f(0.3f, 0.0f, 0.0f, 1.5f);
        glBegin(GL_LINES);
        glVertex3f(0, -2, 0);
        glVertex3f(0, 2, 0);

        //arrow
```

```
        glVertex3f(0.0f, 2.0f, 0.0f);
        glVertex3f(0.1f, 1.8f, 0.0f);
        glVertex3f(0.0f, 2.0f, 0.0f);
        glVertex3f(-0.1f, 1.8f, 0.0f);

        glEnd();

        //z
        glColor4f(0.0f, 0.0f, 0.3f, 1.5f);
        glBegin(GL_LINES);
        glVertex3f(0, 0, -2);
        glVertex3f(0, 0, 2);

        //arrow
        glVertex3f(0.0, 0.0f, -2.0f);
        glVertex3f(0.0, 0.1f, -1.8f);
        glVertex3f(0.0, 0.0f, -2.0f);
        glVertex3f(0.0, -0.1f, -1.8f);

        glEnd();

        glPopMatrix();
    }

    glPopMatrix();
}

void MouseCallback(int button, int state, int x, int y)
{
    mouse = { x, y, button, state };
}

void MouseMotionCallback(int x, int y)
{
    if ((mouse.button == GLUT_LEFT_BUTTON) && (mouse.state == GLUT_DOWN))
    {
        cameraAngleXY.first += (y - mouse.y) * 0.3f;
        cameraAngleXY.second += (x - mouse.x) * 0.3f;
        mouse.x = x;
        mouse.y = y;
    }
    else if ((mouse.button == GLUT_MIDDLE_BUTTON) && (mouse.state ==
   GLUT_DOWN))
    {
        cameraDistance -= (y - mouse.y) * 0.1f;
        mouse.y = y;
    }

    glutPostRedisplay();
}

void KbCallBack(unsigned char key, int x, int y)
{
    switch (key)
    {
    case 's':
        save("test1.pdf");
        glPushMatrix();
        glRotatef(180, 0, 1, 0);
        save("test2.pdf");
        glPopMatrix();
        break;
    }
    glutPostRedisplay();
    std::cout << key;
}

void SpCallBack(int key, int x, int y)
{
    const auto mods = glutGetModifiers();

    if (GLUT_ACTIVE_CTRL & mods)
    {
        switch (key)
        {
        case GLUT_KEY_LEFT:
            cameraTransXY.first -= 0.5;
            break;
```

```
        case GLUT_KEY_RIGHT:
            cameraTransXY.first += 0.5;
            break;
        case GLUT_KEY_DOWN:
            cameraTransXY.second += 0.5;
            break;
        case GLUT_KEY_UP:
            cameraTransXY.second -= 0.5;
            break;
        case GLUT_KEY_PAGE_UP:
            break;

        }
    }
    else
    {

        switch (key)
        {
        case GLUT_KEY_LEFT:
            cameraAngleXY.second -= 0.5;
            break;
        case GLUT_KEY_RIGHT:
            cameraAngleXY.second += 0.5;
            break;
        case GLUT_KEY_DOWN:
            cameraAngleXY.first += 0.5;
            break;
        case GLUT_KEY_UP:
            cameraAngleXY.first -= 0.5;
            break;
        case GLUT_KEY_PAGE_UP:
            break;

        }
    }
    glutPostRedisplay();
    std::cout << key;
}

void ReshapeCallback(int w, int h)
{
    screenWH = { w, h };

    glViewport(0, 0, (GLsizei)screenWH.first, (GLsizei)screenWH.second);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    gluPerspective(45.0f, (float)(screenWH.first) / screenWH.second, 1.0f,
    100.0f); // FOV, AspectRatio, NearClip, FarClip

    // switch to modelview matrix in order to set scene
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void TimerCallback(int millisec)
{
    glutTimerFunc(millisec, TimerCallback, millisec);
    glutPostRedisplay();
}

void MenuCallback(int item)
{
    switch (item)
    {
    case 1:
        drawG = !drawG;
        break;
    case 2:
        drawG2 = !drawG2;
        break;
    case 3:
        drawSphere = !drawSphere;
        break;
    case 4:
        drawFaces = !drawFaces;
        break;
    case 5:
        drawColors = !drawColors;
```

```
        break;
    case 6:
        drawAxes = !drawAxes;
        break;
    case 7:
        drawPoints = !drawPoints;
        break;
    case 8:
        drawFacesSkeletons = !drawFacesSkeletons;
        break;
    default:
        break;
    }

    glutPostRedisplay();
}

int main(int argc, char *argv[])
{
    std::cout << "xyz: ";  std::cin >> fileName;
    std::cout << "colors: ";  std::cin >> nColors;

    points = readXYZ(fileName + ".xyz");
    edges = readTriangEdges(fileName + ".g");
    edges2 = readTriangEdges(fileName + ".g2");
    faces = readFaces(fileName + ".vor");
    coloring = readColoring(fileName + "." + nColors + "c");

    for (auto& f : faces)
    {
        faces2.push_back(upgrade_face(f));
    }

    sedges = std::set<std::pair<int, int>>(edges.begin(), edges.end());
    sedges2 = std::set<std::pair<int, int>>(edges2.begin(), edges2.end());
    std::set_difference(sedges2.begin(), sedges2.end(), sedges.begin(), sedges.end(),
        std::inserter(sedges3, sedges3.end()));

    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH | GLUT_STENCIL);
    glutInitWindowSize(1000, 800);
    glutInitWindowPosition(0, 0);
    glutCreateWindow((fileName + ".xyz").c_str());

    {
        glutDisplayFunc(DisplayCallback);
        glutMouseFunc(MouseCallback);
        glutMotionFunc(MouseMotionCallback);
        glutTimerFunc(100, TimerCallback, 100);
        glutSpecialFunc(SpCallBack);
        glutKeyboardUpFunc(KbCallBack);
        glutReshapeFunc(ReshapeCallback);
    }

    {
        glClearColor(0, 0, 0, 0);
        glShadeModel(GL_SMOOTH);
        glEnable(GL_DEPTH_TEST);
        glEnable(GL_POINT_SMOOTH);
        glEnable(GL_LINE_SMOOTH);
        glEnable(GL_POLYGON_SMOOTH);
        //glEnable(GL_LIGHTING);
        glEnable(GL_BLEND);
        glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
        glHint(GL_LINE_SMOOTH_HINT, GL_NICEST);
        glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
        glHint(GL_LINE_SMOOTH_HINT, GL_NICEST);
        glHint(GL_POLYGON_SMOOTH_HINT, GL_NICEST);
        glHint(GL_POINT_SMOOTH_HINT, GL_NICEST);

        //glPointSize(point_size);
        //glLineWidth(line_width);
    }

    //light
    {
```

```
        GLfloat lightKa[] = { .3f, .3f, .3f, .9f };  // ambient light
        GLfloat lightKd[] = { .7f, .7f, .7f, .9f };  // diffuse light
        GLfloat lightKs[] = { .9f, .9f, .9f, .9f };   // specular light
        glLightfv(GL_LIGHT0, GL_AMBIENT, lightKa);
        glLightfv(GL_LIGHT0, GL_DIFFUSE, lightKd);
        glLightfv(GL_LIGHT0, GL_SPECULAR, lightKs);

        float lightPos[4] = { 0, 0, 1, 0 }; // directional light
        glLightfv(GL_LIGHT0, GL_POSITION, lightPos);
        glEnable(GL_LIGHT0);
    }

    //menu
    {
        glutCreateMenu(MenuCallback);
        glutAddMenuEntry("Show G", 1);
        glutAddMenuEntry("Show G2", 2);
        glutAddMenuEntry("Show sphere", 3);
        glutAddMenuEntry("Show faces", 4);
        glutAddMenuEntry("Show colors", 5);
        glutAddMenuEntry("Show axes", 6);
        glutAddMenuEntry("Show points", 7);
        glutAddMenuEntry("Show faces skeletpons", 8);
        glutAttachMenu(GLUT_RIGHT_BUTTON);
    }

    glutMainLoop();

    return 0;
}
```