Department of Computer Science
Faculty of Mathematics, Physics and Informatics
Comenius University, Bratislava

Cores of uncolourable cubic graphs

Bachelor thesis

2018
Tomáš Vician

Department of Computer Science
Faculty of Mathematics, Physics and Informatics
Comenius University, Bratislava

Cores of uncolourable cubic graphs

Bachelor thesis

Study program:	Informatics
Field:	2508 Informatics
Department:	Department of Computer Science
Advisor:	RNDr. Ján Mazák, PhD.

Bratislava, 2018
Tomáš Vician

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Tomáš Vician
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: Cores of uncolourable cubic graphs
Jadrá nezafarbiteľných kubických grafov

Anotácia: Práca zahŕňa vytvorenie databázy alebo generátora všetkých malých nezafarbiteľných kubických grafov (zhruba do 20 vrcholov) a následne zostavenie databázy minimálnych nezafarbiteľných podgrafov jednak týchto malých snarkov, jednak vybraných netriviálnych väčších snarkov. Zozbierané podgrafy budú podrobené ďalšej analýze; dúfame, že nám to pomôže získať lepší vhľad do štruktúry nezafarbiteľných kubických grafov.

Cieľ:
1. Vytvorenie databázy všetkých malých nezafarbiteľných grafov vrátane triviálnych (napr. s rovnobežnými hranami).
2. Výpočet jadier (minimálnych nezafarbiteľných podgrafov) malých triviálnych i netriviálnych snarkov a ich archivácia v podobe použiteľnej v budúcom výskume.
3. Analýza vypočítaných jadier a odvodenie ich všeobecne platných vlastností.

Vedúci: RNDr. Ján Mazák, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: prof. RNDr. Martin Škoviera, PhD.

Dátum zadania: 24.10.2017

Dátum schválenia: 26.10.2017

doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Tomáš Vician
Study programme:	Computer Science (Single degree study, bachelor I. deg., full time form)
Field of Study:	Computer Science, Informatics
Type of Thesis:	Bachelor´s thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Cores of uncolourable cubic graphs

Annotation: The thesis includes creation of a database or a generator of all small uncolourable cubic graphs (ca. up to 20 vertices) and then computing minimal uncolourable subgraphs of both these graphs and some non-trivial larger known snarks. The computed subgraphs will undergo further investigation; we hope to discover some of their properties that would give new insights into uncolourability of cubic graphs.

Aim: 1. Create a database of all small uncolourable cubic graphs, including the trivial ones (e.g. those containing parallel edges).
2. Compute all cores (smallest uncolourable subgraphs) of small trivial and nontrivial snarks and permanently store them for further research.
3. Analyze the computed cores and derive their common properties.

Supervisor:	RNDr. Ján Mazák, PhD.
Department:	FMFI.KI - Department of Computer Science
Head of department:	prof. RNDr. Martin Škoviera, PhD.
Assigned:	24.10.2017

Approved: 26.10.2017 doc. RNDr. Daniel Olejár, PhD.
Guarantor of Study Programme

.....
Student

.....
Supervisor

Acknowledgements: I would like to thank my advisor RNDr. Ján Mazák for his help, consultations and valuable advices regarding this thesis.
I would also like to thank to my family and friends for all their support.

Abstrakt

Hlavnú kostru práce tvorý program na generovanie nezafarbiteľných kubických grafov (snarkov). K týmto grafom sa ešte pridávajú veľké snarky z už existujúcej databázy. Kedže vytváranie grafov hrubou silou je časovo veľmi neefektívne, musí sa použiť rýchlejší spôsob generovania.Tento spôsob je inšpirovaný teóriou o dekompozícii snarkov. Pre výsledné grafy sa počítajú jadrá, ktoré môžu poslúžiť pri následných overovaniach hypotéz z oblasti teórie grafov. V tejto práci tiež nahliadneme do zložitostných vlastností.

Kľúčové slová: nezafarbiteľný kubický graf, jadro grafu, snark

Abstract

The main frame of this work consists a program for generating uncolourable cubic graphs (snarks). To these graphs we add larger snarks from an already existing database. Since generating graphs with brute force has a large time complexity, a more effective way is needed for creating the graphs. This method is inspired by the theory about the decomposition of snarks. For the resulting graphs we compute their cores, which can be later used for the verification of conjectures from the field of graph theory. In this paper we also gain some insight into complexity properties.

Key words: uncolourable cubic graph, core of a graphs, snark

Contents

Introduction

The aim of this work is to get a database of the cores of uncolourable cubic graph which we take as the definition of simple snarks in this paper. For this we create a generator which produces these graphs and stores them in a database. The program also enables us to compute the cores of larger snarks which we import from an existing database called House of Graphs on the internet. Under the term core we understand a minimal (regarding the edges and nodes) graph that is still uncolourable. It means, that if we delete any of the edges of the core, it becomes colourable. It is also important to note that by colourable we mean that a graphs chromatic index is 4. (i.e. it cannot be properly edge-coloured using 3 colours). Having these graphs in a database we enable others to investigate some of their general properties, verify already existing conjectures, or refute them. In this paper we learn some information about the complexity of this task which also reveals some interesting informations about these types of graphs. In the first chapter we introduce some notation and show how we use the SAT for solving this problem. Then we introduce some details of the program, and some of its technical details. Then we investigate the properties of the generation process itself. Then we say some words about the result storage and draw some conclusions.

Chapter 1

Definitions

1.1 Basic notions

Before dealing with the given problem, we look at the basic definitions and concepts in this chapter. We mention them, since the definitions may differ in some sources. By doing so we can avoid possible misunderstandings when reading this work.

In graph theory graphs are defined as abstract objects specified by a set of vertices (or nodes) V and a set of edges E which are pairs of vertices. Graphs are mostly divided into directed and undirected ones. Undirected graphs are those, where we do not assign a direction to the edges. In other words, we do not distinguish between edges connecting vertices *a,b* based on their order. They are represented as unordered pairs *{a,b}*. We are interested in the undirected graphs when dealing with our problem of uncolourable graphs.

Undirected graph is an ordered pair G = (V, E), where:

- V is the vertex set. It is a non empty finite set of vertices (nodes) of the graph. This set is often denoted *V(G)* or just V.
- E is the edge set. A set of unordered pairs $\{u, v\}$, the edges, where u and v are vertices in V. For the sake of brevity sometimes written as uv. If $u = v$, then we talk about self-loop, or simply loops.

When multiple edges connect two vertices, we talk about multigraphs. In this paper we consider multiple edges having no own identity, meaning that these edges are defined solely by the two vertices they connect. Graph $G_0 = (V_0, E_0)$ is the **subgraph** of graph $G = (V, E)$, if $V_0 \subseteq V$ and $E_0 \subseteq E$. Then we can write $G_0 \subseteq G$.

Let $G = (V, E)$ be a graph, $v \in V$ and $e \in E$. If $e = \{u, v\}$, u and v are called adjacent. If $d = \{u, w\} \in E$, e and d are called incident, also if $v \in e$, e and v are incident.

The **degree of vertex** v is the number of edges incident to it. Also denoted $deg(u)$.
$deg(u) = |\{e \in E \mid u \in e\}|$

Regular graph G is a graph which has all its vertices of the same degree. A regular graph with vertices of degree k is called k-regular. A 3-regular graph is also called cubic.

A u-v **walk** of length n in graph G is an alternating sequence of vertices x_i and edges h_i of the graph $G = x_0, h_1, x_1, h_2, ..., x_{n-1}, h_n, x_n$ where the edge h_i and the vertices x_{i-1}, and also x_i are incident (for i=1,2,...,n), while if h_i is not a loop, then $x_i \neq x_{i-1}$. A walk, where $x_0 = x_n$ is called closed, otherwise it is called open.

A **trail** in graph G is a walk in graph G, such that every edge appears in it at most only once. A trail, where $x_0 = x_n$ is called closed (or simply a circuit), otherwise it is called open.

A **path** in graph $G = (V, E)$ denotes a sequence $P = (v_0, e_1, v_1, ..., e_k, v_k)$ where e_i=$\{v_{i-1}$,$v_i\}$ and $v_{i-1} \neq v_{j-1}$ for i $\neq$ j. With other words the path is a trail, where no vertex is repeated. Its **length** is k.

A **connected graph** G is a graph, where for every vertex v, $u \in G$ exists a path from v to u.

A **cycle** is a sequence of vertices and edges $P = (v_0, e_1, v_1, ..., e_n, v_n)$, where $v_0 = v_n$ and vertices v_0,..., v_{n-1} are mutually distinct vertices of G and for every i = 1,2,...,n is $e_i = \{v_{i-1}, v_i\}$ from E.

A **cycle graph** (or circular graph) denotes a graph, which consists of a single cycle, in other words, vertices connected in a closed chain (a path, which starts and ends in the same vertex).

A graph having a subgraph, which is a cycle graph, is called **cyclic**. Otherwise it is said to be **acyclic**.

A graph $G = \{V, E\}$ is called edge-weighted, if every edge $e \in E$ has a number $w \in R$ assigned to it.

Edge colouring of a graph $G = \{V, E\}$ is a function $c: E \to S$, where $(e_1) \neq (e_2)$ if e_1 and e_2 are parallel edges. The elements of S are called **colours**.

A graph $G = (V,H)$ is said to **be k-edge-colourable**, if we can assign k colours to its edges in a way, when no two incident edges have the same colour. We call a colouring which does not colour incident edges with the same colour acceptable. We often use the therm uncolourable graph in this paper, by which we mean that the given graph is not 3-edge-colourable.

The **chromatic index** of graph G is the smallest natural number k such that the graph G is k-edge-colourable.

Graph G is said to be connected, if for every pair of its vertices, u and v , exists a path starting with the vertex u and ending in vertex v. If the graph is not connected, we call its largest connected parts connected components, or simply components. In a component all the vertices are connected via a path. A connected graph consists of only a single component, which is the graph itself.

A **bridge** (isthmus, cut-edge) of a graph is an edge which does not belong to any cycle in the given graph. The deletion of the bridge increases the number of connected components by one. This follows right from the definition, therefore we do not write the proof here.

A **simple graph**, or strict graph, is an unweighted, undirected graph without self-loops or multiple edges.

Snarks are simple, connected cubic graphs with a chromatic index 4. They are often defined as bridgeless, but we do not require this property strictly as we mention it later in this text. A simple example of a snark is the Petersen graph [pic. 1.1]. We can note here that these graphs may contain multiple edges and self-loops.

The **distance** d(u,v) between vertices u and v of a finite graph is the minimum length of the paths connecting them. If no such path exists (if the vertices lie in different connected components), then the distance is set to ∞ [4].

The length $max_{u,v)}$d(u,v) of the "longest shortest path" between any two graph vertices (u,v) of a graph, where d(u,v) is a **graph distance**. In other words, a graphs diameter is the largest number of vertices which must be traversed in order to travel from one vertex to another [17].

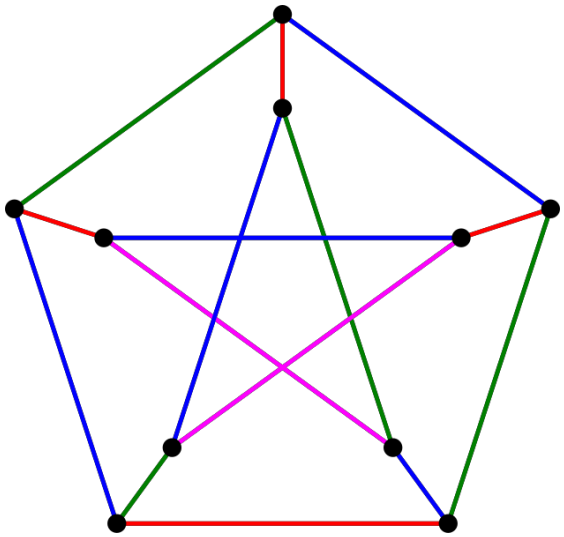

Figure 1.1: Petersen graph [2]

A graph is called **cyclically k-edge-connected**, if minimally k edges must be deleted to disconnect it into two components,where each of them contains a cycle. Such a set of k edges is called a **cyclic k-edge cutset**.

1.2 Transformation to the SAT problem

Valuation of atomic formulas of a language L_p is every function v: P $\rightarrow$ 0,1, which assigns to every atomic formula p $\in$ P a value: 0 (false) or 1 (true).

- With induction on the length of a formula we define the **extension $\overline{v}$ of function** n for the domain of the set of all formulas of language L_p:

$\overline{v}(A) = v(A)$	if A is an atomic formula	
$\overline{v}(\neg A) = 0$	if $\overline{v}(A) = 1$	else $\overline{v}(\neg A) = 1$
$\overline{v}(A \wedge B) = 1$	if $\overline{v}(A) = \overline{v}(B) = 1$	else $\overline{v}(A \wedge B) = 0$
$\overline{v}(A \vee B) = 0$	if $\overline{v}(A) = \overline{v}(B) = 0$	else $\overline{v}(A \vee B) = 1$
$\overline{v}(A \rightarrow) = 0$	if $\overline{v}(A) = 1$ a $\overline{v}(B) = 0$	else $\overline{v}(A \rightarrow B) = 1$
$\overline{v}(A \leftrightarrow B) = 0$	if $\overline{v}(A) = \overline{v}(B)$	else $\overline{v}(A \leftrightarrow B) = 0$

- We say, that $\overline{v}$(A) is the **truth value** of a formula A at valuation v. The formula A is true at valuation v, if $\overline{v}$(A) = 1, otherwise the formula A is false.

The problem of graph colourability is transformable to the SAT problem. It is convenient, because doing so it is enough solve the right SAT problem. With this method we achieve some level of abstraction. In this section follows the description of this process also based on the diploma thesis [10] from our sources.

SAT

On SAT we can look as the set of all satisfiable bool expressions. A formula is in conjunctive normal form (CNF), if it is a conjunction of clause, where every clause is a disjunction of literals. A literal is a variable or its negation. A formula is satisfiable, if exists a valuation of variables such that the formula evaluates to true.

The SAT is a language over the alphabet $\wedge$, $\vee$, $\neg$,(,), 0, 1, where every word belongs to the language, if the corresponding formula in CNF is satisfiable. We can formulate the **SAT-problem** in the following way: There is a given formula with n variables and it has to be decided weather there exists a valuation of variables, when it is satisfied. We can summarize the reason we use a SAT-solver with this quote [9]:

> We are interested in CNF, because the most of modern SAT-solvers works with expressions in CNF. It has the benefit that it is a simple form, where all the clauses has to be satisfied, what is easier to control, and the first algorithms created use this.

It is proved that the SAT problem is NP-complete. The proof for this statement is in Cook-Levin theorem, which is named after Stephen Cook and Leonid Levin, who independently on each other proved the NP-completeness of the SAT problem.

Examples for the SAT problem:

$$\phi_1 = (x_1 \vee x_2) \wedge (x_1) \wedge (\neg\, x_1)$$
$$\phi_2 = (x_1 \vee x_2) \wedge (x_2 \vee x_3)$$

ϕ_1 consists of three clauses (a formula containing literals which evaluates to true, if at least one literal is evaluates to true). The first one has two literals, the second has one and the third one contains the negation of one literal. A formula is not satisfiable, because it contains the conjunction of literals x_1 and its negation. Therefore there exists no valuation at which this formula would evaluate to true.

ϕ_2 is obviously satisfiable. It is enough for x_2 to have valuation 1. If the clauses are restricted to have maximally k literals in one clause, then we talk about k-SAT problem.

DIMACS CNF file format

If we transform our problem to one of a SAT type, then the SAT-evaluator can evaluate, weather the given graph is k-colourable. In our case k = 3.The input format for the SAT solver we use is the DIMACS file format. It allows us to represent boolean formulas in conjunctive normal form as described in [1]. The CNF file format is an ASCII file format.

1. The file may begin with comment lines. The first character of each comment line must be a lower case letter "c". Comment lines typically occur in one section at the beginning of the file, but are allowed to appear throughout the file.

2. The comment lines are followed by the "problem" line. This begins with a lower case "p" followed by a space, followed by the problem type, which for CNF files is "cnf", followed by the number of variables followed by the number of clauses.

3. The remainder of the file contains lines defining the clauses, one by one.

4. A clause is defined by listing the index of each positive literal, and the negative index of each negative literal. Indices are 1-based, and for obvious reasons the index 0 is not allowed.

5. The definition of a clause may extend beyond a single line of text.

6. The definition of a clause is terminated by a final value of "0".

7. The file terminates after the last clause is defined.

p	cnf	2	3
1	2	0	
1	0	0	
-1	0		

Table 1.1: $(x_1 \vee x_2) \wedge (x_1) \wedge (\neg\, x_1)$ in DIMACS CNF format

We can show how is the CNF formula ϕ_1 represented in DIMACS:

For the effective calculation we transform graph colourability into a SAT problem in the following way: the number of variables in the formula is the triple of the number of edges in the graph. The reason for this is that we can assign 3 colours to every edge. Evaluating one of these variables as True means colouring the edge with the colour this variable represents. For every edge we add a clause containing the three possible colours of the given edges. This secures that every edge is coloured with **one colour minimally**. The variables are considered to represent the same colour on different edges if their numeral representations are congruent mod 3. Apart from this, we have to make sure that only one colour can be chosen for a given edge. For this we add three clauses containing all the three possible pairs of the negations of the variables assigned to the given edge. This is done for all the edges and ensures that every edge is coloured with **one colour maximally**. The last thing is to make sure that neighbouring edges have a different colour assigned to them. For this we add clauses containing the disjunction of negations of variables congruent 3 for every incident edge: if e_1 has variables 1, 2, 3 and e_2 has 4, 5, 6, and e_1 is incident to e_2, then we add $(\neg 1 \vee \neg 4) \wedge (\neg 2 \vee \neg 5) \wedge (\neg 3 \vee \neg 6)$. Here we show it on a simple example of graph G, where E_G = (1,2),(1,4),(2,3),(3,4) [1.2].

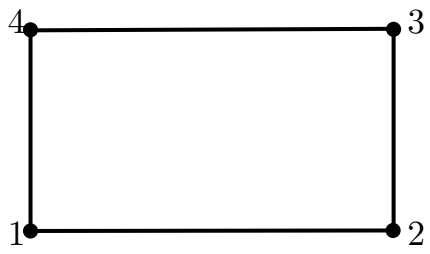

Figure 1.2: Simple sub-cubic graph

p	cnf	12	28					
1	2	3	0					
-2	-3	0	-1	-3	0	-1	-2	0
4	5	6	0					
-5	-6	0	-4	-6	0	-4	-5	0
-1	-4	0	-2	-5	0	-3	-6	0
7	8	9	0					
-8	-9	0	-7	-9	0	-7	-8	0
-1	-7	0	-2	-8	0	-3	-9	0
10	11	12	0					
-11	-12	0	-10	-12	0	-10	-11	0
-7	-10	0	-8	-11	0	-9	-12	0
-4	-10	0	-5	-11	0	-6	-12	0

Table 1.2: DIMACS CNF format for [1.2]

Graph6 Format

The graphs are represented in Graph6 format. The graph is represented as the pair N(n) R(x), where n is the number of vertices and x is the bit vector representing the upper triangle of the adjacency matrix [1.4]. R(x) is the representation of the bytes of x in a decimal form.

Number n is an integer from the interval 0 - 68719476735 (=2^{36}-1).

If 0 <= n <= 62, then N(n) is one byte having the value n+63.

If 63 <= n <= 258047, define N(n) to be the four bytes 126 R(x), where x is the bigendian 18-bit binary form of n.

Examples:
N(30) = 93
N(12345) = N(000011 000000 111001) = 126 66 63 120
N(460175067) = N(000000 011011 011011 011011 011011 011011) =
= 126 126 63 90 90 90 90 90

The vector x is interpreted in the following way: (0/1 on the i-th position in x → vertices x y are connected/unconnected):

The following table shows how are the particular bits assigned to the positions in the vector x for n=4:

i	**x**	**y**
0	0	1
1	0	2
2	1	3
...	...	...
$\frac{n(n-3)-2}{2}$	0	n-1
...	...	...
$\frac{n(n-1)}{2}$	n-1	n-1

Table 1.3: Edges assigned to the bits according to their positions

x/y	**0**	**1**	**2**	**3**
0	.	1	2	4
1	.	.	3	5
2	.	.	.	6
3	.	.	.	.

Table 1.4: Adjacency matrix and the order of edges in x

Example:

Let n=5 and G have 0-2, 0-4, 1-3 a 3-4.

x = 0 10 010 1001

Then N(n) = 68 a R(x) = R(010010 100100) = 81 99.

The resulting encoding is 68 81 99.

Chapter 2

Generating the graphs

In this chapter we are going to describe the process of generating the required cubic graphs. In the first section we talk about the motivation for the construction of this database and its usability. In the next section we describe the main functions used in the program. After that show the main algorithm and we look at the generation of an auxiliary set of graphs used in the main process. This is followed by a brief argument on behalf of the correctness of the whole process. Then we introduce some additional non-trivial snarks by which we enrich our database, mention complexity issues and finish with some technical description.

2.1 Motivation

Cubic graphs constitute an important part of graph theory, especially the snarks. As mentioned in [16]:

"Snarks are quintessential to many important problems and conjectures in graph theory including the 4-colour theorem, Tutte's 5-flow conjecture, the cycle double cover conjecture, and many others. While most of these problems are trivial for 3-edge-colourable graphs, they are exceedingly difficult for snarks in general. On the other hand, for those which are close to being colourable they are usually tractable."

Also described in [7]: "For many of the unsolved problems concerning cycles and matchings in graphs it is known that it is sufficient to prove them for snarks". Many conjectures can be investigated if having a proper database, what is a motivational factor for creating our generator program. As showed in [7], some conjectures can be refuted when having a proper database at disposal.

2.2 Graph functions

Graph generation with some brute force method would have too large complexity so another method is needed. Our approach is therefore to iteratively generate graphs. In this chapter we describe this method. We start the generation from some small cubic graphs and new graphs are generated with the functions we define later in this chapter. Having the process for graph generation it is important to have a proof of correction. We prove that by using these functions we get the set of all uncolourable cubic graphs up to the desired number of vertices. In the process of generation we also use an external graph generator program [13] producing colourable cubic graphs since we need them in our process.

In the section below we describe the functions we use in the process of graph generation. In the description of these functions we use the letters of the English alphabet as vertex labels, in the program however the vertices are labelled with numbers from 1 to n, where n is the number of vertices in the given graph. For the graphs generated by the functions a canonical form is computed, so we can easily avoid generating isomorphic graphs.

- f_1 = **AddTwoNodesOnEdge**
  input: graph G, edge uv, where uv $\in E_G$
  output: graph G'

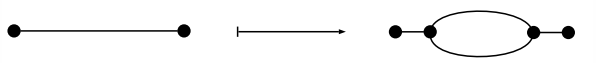

Figure 2.1: Expanding edge with multi edge pair

- f_2 = **addTwoLoopsInsteadEdge** input: graph G, edge uv, where uv $\in E_G$
  output: graph G'

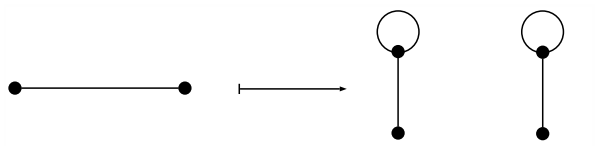

Figure 2.2: Replace edge with edges incident with self-loops

- f_3 = **addTriangle**
  input: graph G, vertex v, where v $\in V_G$
  output: graph G'

- f_4 = **addSquare**
  input: graph G, edges u_1u_2, v_1v_2, where u_1u_2, $v_1v_2 \in E_G$
  output: graph G'

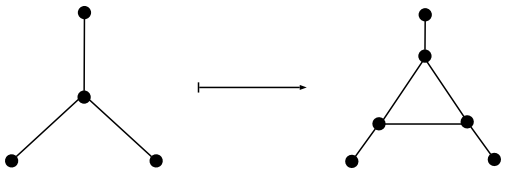

Figure 2.3: Replace vertex with a triangular formation

- f_5 = **joinGraphsOnNodes**
  input: graphs G_1, G_2, vertices v_1, v_2, where $v_1 \in V_{G_1}$ and $v_2 \in V_{G_2}$
  output: graph G'

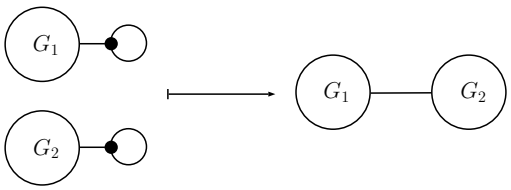

Figure 2.4: 1. version: join graphs on self-loops

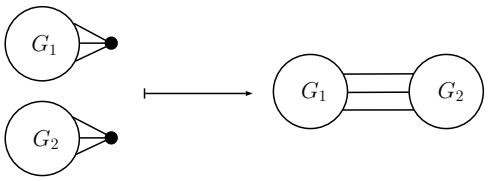

Figure 2.5: 2. version: join graphs on vertices

- f_6 = **joinGraphsOnEdges**
  input: graphs G_1, G_2, edges u_1v_1, u_2v_2,where $u_1v_1 \in E_{G_1}$ and $u_2v_2 \in E_{G_2}$
  output: graph G'

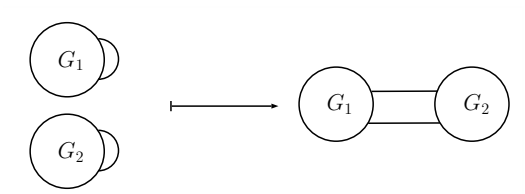

Figure 2.6: Join graphs on edges

- **insertGraph**
  input: graph container S, canonic graph container S_c, graph G
  No return value. Inserts graphs to S they meet the criteria and have no isomorphic equivalents there. Simpler version for colourable graphs in the auxiliary container.

f_1 Adding two vertices on an edge
Adding two vertices on an edge in this function we choose two vertices (v_1v_2) connected with an edge. We remove the edge connecting them and add two other vertices (u_1, u_2) to the graph. Then these additional vertices are connected with the graph in this manner: add edges u_1u_2 two times, v_1u_1 and v_2u_2. In this way the resulting graph is also cubic if the input graph was cubic too. The degree of the vertices v_1 and v_2 stays unchanged since we remove one of their incident edges (v_1v_2), but we also add two other incident edges. Namely v_1u_1 and v_2u_2. We can easily see, that the degree of the added vertices is also 3.

f_2 Adding two vertices with self-loops
An edge v_1u_1 is chosen. It is removed and two vertices, v_2 and u_2 are added to the graph. Then v_1 and u_1 are connected with v_2 and u_2 respectively. Two self-loops, v_2v_2 and u_2u_2 are also added to the graph to preserve 3-regularity of the graph. Before saving the resulting graph whether its connected or not, because it has the potential of creating an unconnected graph if the original graph G contained a bridge. If the resulting graph is not connected, the return value is None.

f_3 Replacing a vertex with a triangle
Two vertices are added to the graph (v_1, v_2). A triangular formation is created by mutually connecting the vertices v, v_1 and v_2 with edges (vv_1), (vv_2),(vv_2). The function then works differently according to three cases:

1. The vertex v is originally connected to three different vertices (u_1, u_2, u_3): the edges vu_1, vu_2, vu_3 are removed and are replaced by new edges (vu_1),(vu_2),(vu_2).

2. The vertex v is originally connected with one other vertex u and E_G contains a loop on v. (vv) is removed and a second edge (v_1v_2) is added to the triangle.

3. v is connected with only one vertex , namely u with three parallel edges edges. Two of them are replaced with edges (v_1u) and (v_2u). This case is relevant only at the start of the generation process, where is G has two vertices connected with a triple edge. In other cases this situation is not present, since a graph can not have more than two vertices, a triple edge and be connected simultaneously.

f_4 Adding a square on two edges

These edges are removed and four new vertices are added (w_1,w_2,w_3,w_4). A square like formation is made by adding edges w_1w_2,w_2w_3,w_3w_4. This square is then connected to the graph with edges v_1w_1, v_2w_2, u_1w_3 and u_2w_4. We can note here that the vertices v_1, v_2, u_1, u_2 are not necessarily distinct.

f_5 Join two graphs on two vertices

Graphs G_1 and G_2 are connected on vertices v_1 and v_2. In this function we distinguish two different cases:

1. The vertices have self-loops. There are two edges in both graphs that contain one of these vertices. In G_1 we have u_1v_1 and (v_1v_1) In G_2 (u_2v_2) and (v_2v_2). We remove the vertices with the loops and connect the two graphs with an edge (u_1u_2). The result is G' containing a bridge.

2. The vertices have no self-loops. The graphs are connected on v_1 and v_2 having 3 neighbours (v_{11}, v_{12}, v_{13} for v_1 and v_{21}, v_{22}, v_{23} for v_2), not necessarily all distinct. Vertices v_1 and v_2 are removed and the graphs are connected with three edges: ($v_1$1, $v_2$1),(v_{12}, v_{22}),(v_{13}, v_{23}).

In some marginal cases this function may return None too. A case like this is when on eof the input graphs is the one having three parallel edges.

f_6 Joining two graphs on an edge pair

The edges (u_1v_1) and (u_2v_2) are removed from the graphs. The graphs are then connected with two edges (v_1v_2) and (u_1u_2). G' is at most 2-connected. If G_1 or G_2 is uncolorable, than G_3 will be uncolorable too. Proof: let the color of the previously mentioned vertices be: c_1 for v_1, d_1 for u_1, c_2 for v_2, d_2 for u_2. When we connect v_1 with v_2, and u_1 with u_2, we can recolour G_2 in a way where the colors c_2, d_2 in G' are replaced with d_1 and c_1 respectively. In case one of the edges is self-loop, None is returned. This is because in this case we would get a graph with a bridge (the vertex having the loop has to be connected to the rest of the graph by one edge only), which we can get in f_5.

insertGraph
This function is responsible for the insertion of graphs into the graph container. When we are inserting graphs to the standard database, they should be uncolourbale. When we use this function for the creation of the auxiliary database, we need to store only the colourable ones. The function has two versions. The one for the auxiliary container does not controll the colourability, because only colourable graphs are generated there. The graph containers (S and S_c) are lists of lists, where on the 0-th position are graphs having two vertices, on the first 4 vertices and so on. From the next steps only the canonic check is used in the simple form:

- At first, the function controls, weather the graph G is not equal to None. This may be the case at scenarios, when the functions responsible for the graph creation return None. For example, when we use f_2 and the graph ceases to be connected, it returns None.

- The next step is the graphs relabelling and the creation of its canonic form. Relabelling means that its vertices are labelled again from 1 to n, where n is the number of vertices of the graph G. The edges of the graphs are changed accordingly. Then the canonic form of the graph is looked up in S_c. If G $\notin S_c$ and G satisfies the colourability requirements, it is inserted into S to a position determined by the size of the graph.

2.3 The algorithm

First of all, we define four containers for the graphs as lists of lists. In one we store the generated graph. In the second one we store the canonic forms derived from the graphs to prevent duplicity. We describe this canonic form later in this chapter in more details. The other two containers are similar to the first one except for that they store the cores of the graphs.

1. The next step we define the initial graph, from which the generator generates the other ones.

   After that we enter the main loop, where i goes from with a pace of two from 0 until (n/2)-1. n is the number of vertices we want to generate and is given as a parameter of the generating function. At each step of the iteration we are generating the graphs from the ones having the number of vertices corresponding to the current value of i. The value 0 corresponds to the first graphs which have 2 vertices, the next ones for i = 1 have 4 vertices and so on.

2. At each level of iteration we apply the functions we described in the previous chapter on the existing graphs. The insertion function provides some checks

before inserting the newly generated graphs. It controls the colourability properties and prevents storing duplicate graphs by searching for the canonic forms of the graphs in the given container. Null pointers are also filtered out since some functions do not necessarily return a graph when not meeting some criteria.

3. The first three functions generate new graphs from those that have less vertices by two. In case of the first two functions, f_1 and f_2 we iterate through all the edges of the current graph and apply both functions on them. Then an iteration is made on all the nodes of the current graph where we apply f_3. Since f_4 increases the vertex number by 4, we have to take all graphs having a vertex number smaller by 4 as the ones currently generated. Having those graphs we iterate through all the edge pairs and apply f_4 on them.

4. The next functions require two graphs as input parameters. Therefore we iterate through all the appropriate graph size combinations when creating the new ones of the desired vertex number. For example, when we want to create graphs having 10 vertices with the function f_5, we take all the graph pairs G_1 and G_2, where the vertex number of G_1 and G_2 is s_1 and s_2 respectively, s_1, $s_2 \in \{(2,10),(4,8),(6,6)\}$. This is because f_5 reduces the overall vertex number by two. When applying f_6, the sum of vertices can be lower by two because f_6 does not delete any vertices. These functions may also generate correct graphs when one of the input graphs is colourable, so we repeat the previous process with the difference that on graph is from the auxiliary set we describe further below. In summary we iterate through every graph set containing graphs with the desired vertex number, then we iterate those sets on the graphs and finally we iterate through the vertices (for f_5) and edges (for f_6) of those graphs.

2.4 Description of auxiliary graph set

During the generation process we need an auxiliary set of every colourable cubic graphs. For this purpose we use an external program called genreg [13] for generating connected k-regular graphs on n vertices. In our case k equals to 3.

This external program generates only simple graphs. In order to get all the colourable graphs we also need graphs with multi edges too. More precisely, graphs containing double edges. For this we use the function f_2 to expand the simple graphs we already have. It is easy to see, that f_2 does not change the colourability properties of a graph. If we insert u_1 and u_2 on an edge v_1v_2 in a way where we get the edges v_1u_1, two times u_1u_2 and u_2v_2, than we can assign the colour of v_1 to u_2 and the colour of v_2 to u_1. After this process the colourability of the graph does not change. A different case is when the initial graph is uncolourable. In this case the colourability of the graphss in

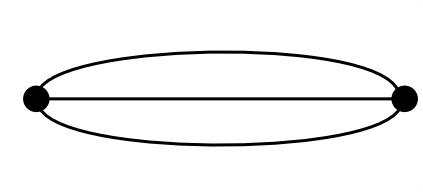

Figure 2.7: Three parallel edges

this auxiliary database is secured by the colourability check in the insertion function made for this database.

To assure the auxiliary database is correct, two main aspects have to be considered. First of all it has to be complete, which means it has to contain all the colourable cubic graphs up to the given degree. Secondly, it should not contain isomorphic graphs for the sake of effectiveness.

The completeness is achieved through the method of generation. The simple graphs are generated and then we iterate through these graphs by their size. The smallest ones are taken one by one and we apply f_2 on all the edges of all the graphs. The resulting graphs are stored in the set of graphs that have two more vertices. During the next step all the simple graphs having two more vertices and the ones generated in the previous step are chosen. Again we apply f_2 on all their edges. This process is repeated until we have the database of graphs up to the required number of vertices. There is one extreme case. It is the graph having a triple multi edge between two vertices [2.7]. No other cubic graph can have a triple edge that is also connected. This is also the only colourable cubic graph with two vertices. Since it is a special case, we just insert it manually to the database before we start the expanding process with f_2. The reason the database is complete is that we generate all the simple graphs, take care of the special case and add the double edges to every graph in all the possible ways.

We can consider isomorphic graphs as same. For this reason we try avoid storing graphs that are isomorphic. This is done by comparing the canonic form of graphs. We also save the canonic formats of graphs during insertion, and if a new graph is being inserted, we search for the same canonic graph as is the one derived from the one being inserted. If we find a same one, we do not save the graph, since we already stored one isomorphic with it. We describe this process later in this paper.

2.5 Proof of correctness

In this section we describe the basic logical concept of this method of generation of graphs. We rely on the "Decompositions and Reductions of Snarks" by Roman Nedela and Martin Škoviera [14]. In this proof we use the term snark as described in the

above mentioned text: "it is convenient to keep the basic notion of a snark as broad as possible. Therefore we allow a cubic graph to have both multiple edges and loops, and understand the term "snark' merely as a synonym for a cubic graph that has no 3-edge-coloring." In addition we are considering only connected graphs. As mentioned in the paper, we can put the operation reducing snarks into two categories: "The first type starts with a snark G, identifies an uncolorable part in it, removes the complement, and replaces it by a small subgraph that is stuck to the remainder to get a snark G' with fewer vertices than G. The resulting snark G' is a reduction of G. The second type of operation splits G into two subgraphs, adds a small number of vertices and edges to each to get two snarks G_1 and G_2, each having a smaller order than G. The pair G_1, G_2 is a decomposition of G." Here we can mention that in case of some decompositions where we get two separate graphs after the decomposition, in some cases one of them can be colourable and for that reason we also generate the auxiliary set of some colourable cubic graphs. We want to generate graphs, therefore we implemented our functions doing the opposite operations as are described in the paper. Our functions f_1, f_2, f_3, f_4 are based on the first type of operations while f_5, f_6 on the second type. When using the later group of functions, we need the set of colourable cubic graphs too, because we need them for the reverse of those reductions, where two graphs are created and one of them is colourable . The proof of that the graphs we create have the desired properties can be divided into more logical steps. The first is their uncolourability and connectedness. These properties are controlled before insertion to the set of graphs. The second step is the proof of completeness, where we need to reason why does our program generate all the graphs cubic graphs we need. This can be done inductively. Here we see from the article, that every uncolorable, cubic, connected graph is generated from a smaller one having similar properties, or two smaller ones with the same properties except that one of those two may be colourable. For the base of induction it is enough to have the smallest cubic graphs, which is G, where E_G = uu, uv, vv [2.8], and a set of all colourable cubic graphs to the desired vertex size. For the inductive step we need all the possible ways in which a graph with correct properties is generated from the previous graphs. These are implemented in our functions that are based on the decomposition operations described in article [14]. As described in 2.3 at number [3.] and [4.], the generating functions are applied on every graph or graph-pair, on every vertex, edge or edge pair according to their specification. This ensures that every graph is generated that could be with the methods based on [14]. After the process the resulting database is expanded by cyclically 4-connected graphs with diameter 5 from [TODO -source].

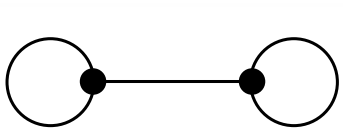

Figure 2.8: Smallest uncolourable cubic graph

2.6 Adding larger snarks

To have larger snarks that are non-trivial, we include some from a web database of graphs called House of graphs [6]. We import them up to 20 vertices. There is nine of them in total. The first is the Petersens graph which has 10 vertices. There are 2 graphs in this set having 18 vertices and 6 with 20 vertices. On the current hardware the core decomposition has finished only for the Petersens graph. Some statistical details are mentioned in a later chapter of this paper. On the previously mentioned website the graphs were in Graph6 format, which we parsed with the NetworkX Python library used in our program.

2.7 Complexity of the program

During the computation of the complexity of the program, we can proceed in three steps:

1. At first we can look at the complexity of our auxiliary programs used during the graph generation. One of these programs is "nauty", which uses a SAT solver called "lingeling" [5]. We know that the SAT-problem is is NP-complete (nondeterministic polynomial time) [15]. As well as the program for creating canonical labels for graphs works with NP complexity. The formal definition of NP is the set of all decision problems that can be solved in polynomial time by a non-deterministic Turing machine.

2. The next issue is the set of functions used in the program. The first are f_1 and f_2 which have linear complexity depending on the number of edges in the graph, denoted O(E). The amount how many times is f_3 called on a graph depends on the number of its vertices - denoted O(V). f_5 operates with two graphs on their nodes, therefore the complexity class is O(V^2). f_6 is similar, where the induced complexity is O(E^2), since it operates on the edges of the graphs.

3. The last, but the most important factor is the number of generated graphs in the previous iteration in every round of iteration. Here we prove it having a

lower boundary. As we described previously, some uncolourable graphs can be generated from on colourable and one uncolourable with a specific operation. If we take a core of the Peterson graph, we can join it with any colourable graph, what implies an overall exponential complexity.

Some numeric results about the generation:
2 nodes -> 1 graph -> added manually
4 nodes -> 3 graphs -> 0.16416 seconds
6 nodes -> 15 graphs -> 1.53660 seconds
8 nodes -> 74 graphs -> 14.60413 seconds
10 nodes -> 436 graphs -> 150.02060 seconds

2.8 Technical features

Python 3
In this section we mention some of the technical detail of the program generating uncolourable cubic graphs. The main part of the program was written in Python 3 [3], but used some external programs written in c programming language too. The Python part of the program uses mainly the library called NetworkX [8]. It has implementations of many structures and functions regarding graph theory, which provides a comfortable framework for a user working with graphs.

Nauty
Nauty and Traces [12] are programs written in C for computing automorphism groups of graphs. They can also produce canonical labels for graphs, what we use in our program. The input is a simple graph in Graph6 format and the output is the canonical label for the given graph.

There is a small suite of programs called gtools included in the package. Another program, created by Ján Mazák [11], also using gtools, provides functions for controlling graph colourability. Our generator uses it too for checking colourability properties. It transforms the colourability problem into a SAT problem and solves it with a program called Lingeling [5]. The C functions from these programs we use are accessed via the ctypes Python library. Ctypes allows the user calling functions in DLLs or shared libraries of C programs.

Genreg
Genreg is a graph generator [13] which we used for the generation of simple cubic graphs for the auxiliary database.

Chapter 3

Reduction to cores (minimal uncolourable subgraphs)

We define the core of a graph (a snark) as a minimal sub-graph which is still uncolourable. It means that removing an edge from the graph makes colourable (reduces its chromatic index).

3.1 Description of the algorithm producing cores

The input of the function generating the cores of a graph is a container where are the cores stored, another container for the canonic labels of the graph for evade redundancy and a graph G. We remove those nodes from G which have a self-loop on them, because loops are trivial cores (they are uncolourable). Then we try to remove every edge recursively (a recursion on all edges). If a graph is uncolourable and all its sub-graphs are colourable, we save the given graph. If it has uncolourable sub-graphs, we continue the recursion on them. If the graph is colourable, we return from the recursion. This implies exponentioal complexity in terms of the number of edges. However, the recursion is usually 4 calls deep, what makes it feasible to compute the cores for a reasonable amount of graphs as we show some numeric results later.

3.2 Possible improvement

To make the program more effective, we could use some more low level programming , like C, A better synchronisation of the used software could also reduce the time needed for the generation. Some hardware with bigger capacity could help us to generate a bigger database as well. We could also replace the SAT solver with some back-track algorithm unless we do not exceed some specific vertex number in the graph.

Chapter 4

Analysis of the results

4.1 Statistical data

In this chapter we show some statistical data resulting from the generation of the graph.

Vertex number	amount of graphs	time needed
2	1	added manually
4	3	0.16416072845458984 seconds
6	15	1.5365965366363525 seconds
8	74	14.604133605957031 seconds
10	436	150.02060294151306 seconds

Table 4.1: Basic generation of small snarks

Vertex number of original graphs	amount of cores	time needed
2	0	added manually
4	1	0.16416072845458984 seconds
6	6	1.5365965366363525 seconds

Table 4.2: Generating cores

The Petersen graph is the first large snark with 10 vertices. It has 3 non-isomorphic cores: 1 with 8 vertices and 2 with 10 vertices. The measured computation time for these cores is 75.59278 seconds.

Chapter 5

Storage of the results

In this chapter we are describing how the graphs are stored. After that we show a possible method to reduce the required storage space for cubic graph by creating a transformation to simple graphs.

5.1 Format of the stored graphs

The graphs, when generated, are stored in a list of lists by size. In the main list, on the first position with index 0, are the smallest possible cubic graphs having only two vertices. With increasing index in the graph container, increases the number of vertices of the graphs on that given position. At the k-th index is the list of graphs having 2k+2 vertices. When we insert a graph to the container, the "insertGraph" function appends the new graph to the end of the proper list. Since the number of generated graphs is relatively small, they are stored in a text file, each of them represented by their edges written as pairs of numbers, where the numbers represent the vertices. An example for a graph G, where $V_G = \{1,2,3\}$, $E_G = \{12, 12, 23, 33\}$, the text representation of this graph will be [(1, 2), (1, 2), (2, 3), (3, 3)].

5.2 Cubic to simple transformation

In case we would have to deal with significantly more cubic graphs, we could represent them in Graph6 format too. As we know, Graph6 format does not support multi-graphs, however cubic graphs can be transformed to simple graphs bijectively (except for the ones having two vertices as we show later). The problematic parts are the dual edges and self-loops. The process of transformation of cubic graphs to simple ones is the following: removing self-loops and replacing parallel edges with single ones. The transformation in the opposite direction is a bit more complicated. If we encounter an edge that is connected to the rest of the graph only at one of its vertices, we put

a self-loop on its other vertex, because no other way exists to expand the graph to make the loose node cubic. Here we can note, that a self-loop increases a degree of a vertex, it is incident with, by two. A more complex case is when we have an edge e, that is connected to the graph with both of its vertices which are not cubic. In this case there are two possibilities. Either we have to insert to this graph a parallel edge with e, or a parallel to one of its incident edges. Lets call a path consisting only such ambiguous edges an ambiguity chain. Let us consider only those ambiguity chains that are maximal regarding their length (their first and last edge is incident with a non "ambiguous" edge). Since we can not add a parallel edge to those incident with the list (because one of their vertices is already cubic), we have to add it to the next edge. In this way the chain gets shorter by two edges (unless it was the last edge in the chain). It is important to note here, that such chains always have an odd number of edges. With this method we manage to retrieve the original form of the graph. Another case is when the ambiguous edges are in a cycle. It this situation there is an even number of edges and two possibilities for expanding such a cycle. This, however, is not a problem, since the two possible resulting graphs of the expansion are isomorphic. Here we can avoid this ambiguity by always adding the parallel edge to the first possible place. By first we mean edge uv, where the sum of the numerical representations of u and v is minimal.

During the conversion of cubic graphs to simple ones ambiguity occurs when dealing with graphs having two vertices as mentioned before. This is because transformation from cubic graphs produces the same result for both graphs with two vertices: [2.8] and [2.7]. Therefore we can handle this situation as a special case and save storage capacity on every other graph.

Conclusion

We gained some insight into the generation of small snarks in this paper. We see that there are some complexity issues that make the process slower. We also see some room for improvement, which could increase the feasible amount of graphs we can generate. Future research may rely on some of these experiences.

Bibliography

[1] Cnf files. http://people.sc.fsu.edu/~jburkardt/data/cnf/cnf.html. (Online; accessed 15-May-2018).

[2] Petersen graph. https://en.wikipedia.org/wiki/Petersen_graph. (Online; accessed 15-May-2018).

[3] *Python Software Foundation. Python Language Reference*, version 3.4 edition.

[4] Barile, Margherita. "from mathworld–a wolfram web resource, created by eric w. weisstein. http://mathworld.wolfram.com/GraphDistance.html.

[5] Armin Biere. Lingeling, plingeling and treengeling entering the sat competition 2013.

[6] J. Goedgebeur H. Mélot Brinkmann, K. Coolsaet. House of graphs: a database of interesting graphs, discrete applied mathematics. *Journal of Graph Theory*, 161:311–314, 2013. (Available at http://hog.grinvin.org).

[7] JonasHägglund Klas Markström Gunnar Brinkmann, Jan Goedgebeur. Generation and properties of snarks. 103:468–488, July 2013.

[8] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008.

[9] Matúš Kukan. Využitie sat-solverov pri riešení ťažkých úloh. Bakalárska práca, Univerzita Komenského v Bratislave, 2017.

[10] Olívia Kunertová. Circular chromatic index of small snarks. Diplomová práca, Univerzita Komenského v Bratislave, 2017.

[11] Ján Mazák. personal communication, 2017-2018.

[12] Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, {II}. *Journal of Symbolic Computation*, 60(0):94 – 112, 2014.

[13] M. Meringer. Fast generation of regular graphs and construction of cages. 30:137–146, 1999.

[14] Roman Nedela and Martin Škoviera. Decompositions and reductions of snarks. *Journal of Graph Theory*, 22(No. 3):253–279, 1996.

[15] Ohrimenko, Olga; Stuckey, Peter J.; Codish, Michael. "propagation = lazy clause generation", principles and practice of constraint programming. Lecture Notes in Computer Science, 2007.

[16] Ján Mazák Martin Škoviera Robert Lukoťka, Edita Máčcajová. Small snarks with large oddness. 22:2, 2015.

[17] Weisstein, Eric W. "graph diameter." from mathworld–a wolfram web resource. `http://mathworld.wolfram.com/GraphDiameter.html`.