

# ZADÁNÍ BAKALÁŘSKÉ PRÁCE

## I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení:	Ullrich	Jméno: Herbert	Osobní číslo: 434653
Fakulta/ústav:	Fakulta elektrotechnická		
Zadávající katedr	a/ústav: Katedra počítačů		
Studijní program:	Otevřená informatika		
Studijní obor:	Softwarové systémy		

## II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

### Uživatelsky rozšiřovatelná grafová databáze

Název bakalářské práce anglicky:

### User Extensible Graph Database

### Pokyny pro vypracování:

Design and implement an application capable of adding into a graph database such graphs which are not structural duplicates of any other graph in the database. The problem of duplicates is equivalent to the problem of graph isomorpism which is not theoretically solved yet. Examine and evaluate available SW tools which can be used for practical isomorphism verification in a large collection containing at least tens of millions of graphs. Select an appropriate tool and demonstrate its range of applicability in database graphs.felk.cvut.cz.

Design and at least partially implement a system capable of maintaining in the database an information whether various important classes of graphs are completely stored in the database. Typically, such classes are characterized by a combination of a few graph properties, like order, size, regularity, connectivity, etc. The functionality of the system might not be entirely automated, an occasional intervention of graph expert would be an acceptable feature of the system. Your applications are to be accessible through the web interface created by other authors. Provide an appropriate programmer documentation of your project.

Seznam doporučené literatury:

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein: Introduction to Algorithms, 3rd ed., MIT Press, 2009

- [2] J. Matoušek, J. Nešetřil: Kapitoly z diskrétní matematiky, Karolinum, 2010
- [3] R. Sedgewick: Algorithms in C Part 5: Graph Algorithms (3rd Edition), Addison-Wesley Professional, 2002
- [4] J. Demel: Grafy a jejich aplikace, Praha, Academia, 2002

Jméno a pracoviště vedoucí(ho) bakalářské práce:

RNDr. Marko Genyk-Berezovskyj, katedra kybernetiky FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: 31.01.2018

Termín odevzdání bakalářské práce: 25.05.2018

Platnost zadání bakalářské práce: 30.09.2019

RNDr. Marko Genyk-Berezovskyj podpis vedoucí(ho) práce podpis vedoucí(ho) ústavu/katedry

prof. Ing. Pavel Ripka, CSc. podpis děkana(ky)

# III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

Datum převzetí zadání

Podpis studenta



CZECH TECHNICAL UNIVERSITY IN PRAGUE



Faculty of Electrical Engineering Department of Computer Science

**Bachelor's Thesis** 

# User extensible graph database

Preventing duplicates and deciding class completeness over a large set of graphs

Herbert Ullrich Open informatics - Software Engineering

May 2018 http://graphs.felk.cvut.cz Supervisor: RNDr. Marko Genyk-Berezovskyj

# Acknowledgement / Declaration

I would like to thank to:

RNDr. Marko Genyk-Berezovskyj for all the good advice and optimism he gave me.

My girlfriend, Elena Lecce, for maintaining patience with me throughout the writing process.

Petr Olšák for the CTUstyle2 template and to the developers of GraphViz graph visualisation, that greatly improved the aesthetic look of the thesis. I declare that I have written the bachelor thesis completely by myself and that all sources and means are properly cited.

In Prague, May  $25^{th}$ , 2018

# Abstrakt / Abstract

Tento dokument zkoumá dostupné algoritmy a software použitelný pro zabránění výskytu isomorfů (strukturních duplikátů) v početné a dále rozšiřitelné množině grafů a pro rozhodování, zda je specifická třída grafů v databázi přítomna kompletně.

Pro tyto problémy navrhuje dvě praktická řešení – kanonické značení grafů v databázi pomocí **nauty** a ad-hoc algoritmus "count all – find superclass" pro poloautomatické rozhodování. Obě implementuje v podobě příspěvku do projektu Web Graph Service na graphs. felk.cvut.cz

**Klíčová slova:** grafy; isomorfismus grafů; nauty; kanonické značení; třídy grafů; kompletnost dat

This document investigates specific algorithms and software available, that could be used for maintaining a large and extensible set of graphs without isomorphs (structural duplicates) and for deciding whether it contains some specific classes of graphs in their entirety.

Furthermore, it suggests two practical solutions for these problems – **nauty** canonical labeling of the database and a "count all – find superclass" semiautomated decisioning ad hoc algorithm, both of which were contributed to the project of the Web Graph Service on graphs.felk.cvut.cz

**Keywords:** graphs; graph isomorphism; nauty; canonical labeling, graph classes; data completeness

# **Contents** /

1 In	troduction1
1.1	Graph database1
1.2	Examples of graph databases1
1.3	Web Graph Service2
1.4	Motivation2
2 Gr	aph isomorphism $\ldots \ldots 4$
2.1	Isomorphism detection4
2.2	Isomorphisms in a graph
	database4
	2.2.1 On-insert isomorphism
	check5
	2.2.2 Available software for
	isomorphism check6
3 Gr	aph canonization7
3.1	Software for graph canoniza-
	tion7
3.2	graph6 format8
	3.2.1 Format description8
	3.2.2 Representing a graph
	using a graph6 format8
	3.2.3 Parsing graph6 format9
3.3	Canonized graph database9
	3.3.1 Benchmarks 10
<b>4</b> Cl	ass completeness $\dots \dots \dots 11$
4.1	Loose formal definition of
	the problem 11
4.2	List of possible approaches 11
4.3	Relaxations to the counting
	approach 12
	4.3.1 Simulating the oracle
4.4	$access \dots 12$
4.4	Count all – Find superclass
	algorithm 13
	4.4.1 Subclass – superclass
	principles 13
	4.4.2 Count an
E la	4.4.3 Find superclass
5 IN 5 1	Compute properties and in
0.1	compute properties and m-
59	"Count all – Find super-
0.2	class" graph counter 18
	5.2.1 Configuring the class
	inclusion rules through
	the <b>Python API</b> 10
	$\mathbf{u} \mathbf{u} \mathbf{u} \mathbf{v} \mathbf{u} \mathbf{v} \mathbf{u} \mathbf{u} \mathbf{u} \mathbf{u} \mathbf{u} \mathbf{u} \mathbf{u} u$

	5.2.2 Count all — no NULLS	
	heuristics	20
6.3	Recommendations	22
Re	eferences	24
A Da	ata used for visualisations	27
A.1	Batch canonical labeling	
	with nauty and Traces	27
A.2	Adding heuristics to the	
	"count all" procedure	28
B Co	ontents of the attached CD	29
C GI	ossary	30

# **Tables / Figures**

3.1.	Examples of graph6 graph
	representation9
4.1.	Sizes of graph classes on $n$
	vertices 13

<b>2.1.</b> Three drawings of graph $K_{3,3}$			
3.1.	. Canonical labeling of a single		
	graph benchmarks7		
3.2.	Example of conversion to		
	graph68		
3.3.	Comparison of <b>nauty</b> and		
	Traces 10		
5.1.	Entity diagram of graph WGS		
	counter 18		
5.2.	Count all — "No NULLS"		
	heuristics performance impact . $21$		

# Chapter **1** Introduction

There are many collections of graphs available on-line, some of which contain as much as hundreds of millions of them, such as [1] and [2].

Large sets of graphs could be used for various academic purposes, such as finding a challenge for graph algorithms, counting or even enumerating all the graphs of given class. Their use is, however, limited by their unsortedness. One can not easily query graphs matching a specific conditions without having to parse the graph files programatically as they are typically not in a *human-readable* format.

## 1.1 Graph database

**Construction 1.1.** graph database<sup>1</sup> Let's have a quadruple  $D = (\mathbb{G}, f, R, m)$  where  $m \in \mathbb{Z}$ ,  $\mathbb{G}$  is a finite set of simple graphs,  $f = (f_1, \ldots, f_m)$  is an *m*-tuple of functions,  $R = (R_1, \ldots, R_m)$  is an *m*-tuple of sets and  $f_i : \mathbb{G} \mapsto R_i$  for  $i \in \{1, \ldots, m\}$ . We call D a graph database if:

1. There exists a function query from set of all possible *m*-tuples  $v = (v_1, \ldots, v_m), v_i \in R_i$  to the set of all subsets of  $\mathbb{G}$ , such that for every graph  $G \in \mathbb{G}$ :

$$G \in query(v_1, \ldots, v_m) \iff \forall i \in \{1, \ldots, m\} : v_i = f_i(G)$$

2. For any  $G \in \mathbb{G}$  values  $f_1(G), \ldots, f_m(G)$  can be enumerated in  $\Theta(m|\mathbb{G}|)$  or  $\Theta(m \cdot log|\mathbb{G}|)$ *i. e. all the values are precomputed and stored in a (possibly ordered) tabular structure having one line for every*  $G \in \mathbb{G}$ 

We then introduce the terms width of the database, denoted as w(D) = m and size of the database, denoted  $|D| = |\mathbb{G}|$ 

## 1.2 Examples of graph databases

All the following statements on the size, width and other properties of the databases are actual to the date May 20. 2018

**House of Graphs** [4] lists 13,343 graphs it claims to be *interesting*. There are 33 graph properties (*invariants*) listed for every graph, that are not always fully computed. Graphs are identified by their adjacency matrix and — in a user-friendly manner — a picture. This, however, makes it hard for the database to be accessed programatically (taking in account, that the adjacency matrices exceeding certain dimension are not being displayed in its interface).

<sup>&</sup>lt;sup>1</sup> We use the term of graph database in the sense of "database that contains graphs". In other literature graph database can also mean a database engine that "stores the data in a graph structures", such as **neo4j** [3]. This kind of graph databases is nowhere else referred to, throughout the thesis, to avoid the possible misinterpretation.

Introduction

**Encyclopedia of Graphs** [5] offers 23 collections of a combined size of 2,949,675 graphs<sup>1</sup>. Note that the collections *overlap* (e. g. every *strongly regular graph* is also a *regular graph*), hence the actual size of the database is smaller. The visualisations of the graphs are generated from their **sparse6** encoding, that is stored on its server. The *user interface* emphasizes browsing a single *collection* of graphs having some common properties, not giving user the access to the *entire* data. The data is exportable in **CSV** format, however, the graphs can not be easily parsed from it, as they are, on output, represented by an *identifier* rather than a graph encoding.

**ToTo Tree Width Webservice** [6] is a graph database with a narrow focus on the *tree* width graph property, which can be used to decide other graph properties in a fast manner. It implements several interesting ideas, such as distributing the workload of the *tree width* computation to its clients. The database is accessible through an open API and contains the graph6 encodings of the graph. The size of the database is not public, its width is limited to 7 properties related to *tree width*.

## 1.3 Web Graph Service

The idea behind the project is to build a database *bigger* and *wider* that any of the previously mentioned. A database to contain at least *tens of millions* of graphs, with an emphasis on the *completeness* of the graph classes stored in it. Generally speaking, it could provide a supportive environment for the novel algorithms, matching them against the *complete* sets of possible input and enable the *quantitative* research approach over its giant data set, which could be then used as a kind of a *graph corpus*.

The Web Graph Service (hereinafter referred to as WGS), strives to be the biggest open *graph database* offering millions of graphs queriable by their properties, such as a specific order or chromatic number, through an easy-to-use webpage and an API.

The project was assembled with a size of ~  $19 \cdot 10^6$  graphs and introduced in June 2017 by the author of this thesis, Herbert Ullrich and RNDr. Marko Genyk-Berezovskyj. The source and look of **WGS** and its maintenance tools preceding this thesis (and the ones of Tomáš Roun [7] and Sergej Kurbanov [8]), was preserved on [9] and [10] respectively.

The ongoing further research is desirable, as (of June 2017):

- more graph properties could be supported
- user experience could be enhanced by replacing the plain HTML user interface with a more aesthetical one
- maintenance tools written in Java perform well, but do not integrate a third party software that would make computing the graph properties easy for the administration
- user is a mere *consumer* of the data, who can not participate in *collecting* it
- there is not a clear statement on the *class completeness* of the database (like the one on [5]), so the user can not easily know if the data returned to him is complete or partial

# 1.4 Motivation

The possibility of users **adding graphs into the database** is very valuable way of involving the end user in any *graph database*. It can both provide an interesting data for the database and prevent a possible user's frustration over a missing data.

 $<sup>^{1}</sup>$  As the database does not make an explicit statement on its size, this number was mined using a JavaScript snippet browsing through the pages of http://atlas.gregas.eu/collections

However, in order to maintain the integrity of the database and to prevent abuse, several problems have to be dealt with, the hardest of them being the graph isomor**phism problem** (GI). Graph database, that would *prevent* the user from inserting any graph that is isomorphic to any graph already in the database, would keep its space complexity reasonable over time and ensure never listing the same graph twice.

Maintaining an information on whether the queried graph classes are completely stored in the database would raise its sovereignty, by having means to show the user if the data output to him is complete. If it is, the user would have to search no more and go straight to utilizing the queried data for his or her purpose. As we are practically talking about classes of graphs given by  $\sim 40$  properties, each having at least two possible values, the full manual configuration is not an option. The reasoning should therefore be **semiautomated**.

The following chapters aim to give practical solutions for the both aformentioned problems and to explain their theoretical background. Both the solutions were already contributed to the project of WGS and are to be seen working live on http://graphs. felk.cvut.cz (as of May 2018)

# Chapter **2** Graph isomorphism

**Definition 2.1.** An **isomorphism** of graphs  $G_1$  and  $G_2$  is a bijection between the vertex sets of  $G_1$  and  $G_2$ 

$$f: V(G_1) \mapsto V(G_2)$$

such that any two vertices u and v of  $G_1$  are adjacent if and only if f(u) and f(v) are adjacent in  $G_2$ . If an isomorphism exists between two graphs, then the graphs are called **isomorphic** and denoted as  $G_1 \simeq G_2$ . In the case when the function is a mapping of a graph onto itself, i.e., when  $G_1$  and  $G_2$  are one and the same graph, the function is called an **automorphism** of  $G_2$  [11].



**Figure 2.1.** Three drawings of graph  $K_{3,3}$ . Every function  $f: V(G_x) \mapsto V(G_y)$  mapping a vertex of  $G_x$  to a vertex of  $G_y$  with the same color is an isomorphism of  $G_x$  and  $G_y$ .  $(x, y \in \{1, 2, 3\})$ 

# 2.1 Isomorphism detection

Isomorphism detection is an example of a problem with an unknown complexity [12], as there are many algorithms known efficient for a specific graph classes, but there is no general polynomial time algorithm discovered yet. In his recent paper, László Babai claimed the graph isomorphism problem to be quasipolynomial time complex [13]. The claim has been first withdrawn, then restored in January 2017 [14] as a mistake was found in the analysis and as it was modified.

## 2.2 Isomorphisms in a graph database

There are several ways of handling isomorphisms in a graph database D

• Storing every graph in its every structural rotation would, to put it bluntly, bloat the database through the roof. It would suddenly become very difficult to store and maintain a complete collection of graphs in all their isomorphs even on a small graph

order. For example, there are 268,435,456 labeled graphs having an order of 8 [15]. Knowing that there are only 12,346 unlabeled graphs of the order 8 [16], we are dealing with an overhead of 268,423,110 graphs sharing all the attributes with some other graph in D.

**Relying on the isomorphic graphs not being inserted** is highly risky. A Graph database, once populated, might become a rich source of information providing e.g. a count of graphs having specific attributes or a complete list of them.

Whereas the sequences counting labeled graphs satisfying a certain condition on n nodes can often be expressed through combinatorics the problem of expressing the number of unlabeled graphs on n nodes typically requires building all the graphs matching and counting them. Therefore a database containing a complete collection of graphs of a given order would carry valuable data useful to enumerate and verify these sequences.

Having a single isomorph inserted in the database by accident or a malicious user would corrupt this entire data.

• Asserting the non-isomorphism of the stored graphs could be very expensive, as it could require lengthy attribute checks to determine if the graph being inserted is not present in the database yet, in any possible *structural rotation*. Even with the possibility of performance impact, asserting the non-isomorphism of all graphs in database is still the most efficient way of minimizing its space complexity and maximizing its informational value.

### 2.2.1 On-insert isomorphism check

**Theorem 2.1.** Let  $G_1 = (V_{G_1}, E_{G_1})$  and  $G_2 = (V_{G_2}, E_{G_2})$  be isomorphic graphs. Then they have the same number of vertices and the same number of edges [17]

**Theorem 2.2.** Let  $f : G_1 \to G_2$  be a graph isomorphism and let  $v \in V_{G_1}$ . Then deg(f(v)) = deg(v) [17]

**Corollary 2.3.** Let  $G_1$  and  $G_2$  be isomorphic graphs. Then they have the same degree sequence. [17]

From that, we can easily design an algorithm like the one loosely outlined in 2.2.1 (note that it can be easily expanded for the general case of D with m columns, possibly containing NULL property values, by querying them aswell).

The **time complexity** of algorithm 2.2.1 is proportional to  $O(|D|) \cdot T(G \simeq H)$ , as it iterates through O(|D|) graphs and checks the isomorphism against each.  $(T(G \simeq H)$  denotes the complexity of deciding the graph isomorphism between G and H.)

Note that this differs from the naïve approach only by filtering out the unnecessary checks against graphs that were precomputed with at least one different significant attribute  $(f_1, \ldots, f_4)$ .

Performance for inserting multiple graphs could be enhanced e. g. by querying all the suspected graphs in the database at once and then sorting them (and the input) by their properties. Therefore, the query for every one of them would be saved, while still matching every input against the same, correct, number of graphs.

Advantages of this approach are that it assumes nothing about the format of graphs stored in D and that the assertion of equal *significant properties* usually filters out the vast majority of graphs in D, making the running time feasible. Having a D populated only with graphs that passed the check in 2.2.1 would *ensure* the absence of isomorphisms in D.

**Disadvantage of this approach** is the performance in the worst case (e. g. groups of strongly regular graphs sharing all the significant properties). Also making O(|D|)

### Input:

- 1. Graph database  $D = (\mathbb{G}, f, R, m)$  such that m = 4 and  $\forall G \in \mathbb{G}, G =$ (V, E):
  - (a)  $f_1(G) = |V|$
  - (b)  $f_2(G) = |E|$
  - (c)  $f_3(G) =$  "degree sequence of G"
  - (d)  $f_4(G) = |$  "connected components of G"

i. e. a database table of five columns including the graph representation, with all the values necessarily precomputed

- 2. Graph  $H = (V_H, E_H)$
- 3. Function query as from construction 1.1 (invariant 1 holds  $\forall G \in \mathbb{G}$ )

**Output true** if H is *isomorphic* to any graph in  $\mathbb{G}$ , false otherwise 1: procedure SHOULDBEINSERTED

 $h \leftarrow (|V_H|, |E_H|, |\text{``degree sequence of } H''|, |\text{`` con. components of } H''|)$ 2: 3:  $\mathbb{H} \leftarrow query(h_1, h_2, h_3, h_4)$ 4: for all  $G \in \mathbb{H}$  do if  $G \simeq H$  then 5: return true 6: return false 7:

Algorithm 2.2.1. On-insert isomorphism check

isomorphism evaluations on every insert makes it not suitable to perform a *batch in*sertion.

#### Available software for isomorphism check 2.2.2

Even though the graph isomorphism problem (GI) is being argued to be solvable in a quasipolynomial time  $exp((log n)^{O(1)})$  [13], the exponential worst-case complexity software is still being used as the most common practice.

However the algorithms listed below tend to detect the isomorphism in a very fast manner with their worst cases being rare. All the competitive algorithms listed below are based on building a search tree of graph colourings.

- **nauty** created by Brendan D. McKay [18] [19] with a key concept of *partition (colour*ing) refinement, decides the isomorphism by searching for the graph's automorphism group or canonical labeling building a search tree of its possible colourings (with an empty colouring being its root) depth-first, keeping its lexicographically smallest leaf and pruning wherever possible [20]. It is still being likely the most used and one of the fastest competitive programs for deciding the GI problem (since its introduction in 1981).
- **Traces** by Adolfo Piperno [19] introduces a different order of building the tree, as a variant of breadth-first order [20]. With the breadth-first order's obvious disadvantage in not knowing the leaves, that would enable the pruning, in time, it first builds an "experimental path" for every node in the graph in order to find some.
- **saucy** by Paul T. Darga and others [21] reimplemented the original **nauty** algorithm from [18] with several new ideas, running very fast on large and sparse graphs.
- **Bliss** by Tommi Juntilla and Petteri Kaski [22] also based around the original nauty idea, leading it to a fast sollution for some originally difficult graphs.

# Chapter **3** Graph canonization

**Definition 3.1.** a canonical form (also referred to as a canonical labeling) of graph G is a labeled graph Canon(G) that is isomorphic to G, such that every graph that is isomorphic with G has the same canonical form as G [23]

Canonical labeling thus enables an efficient test of graph isomorphism, as it is easier to check the equality of two graphs than their isomorphism.

## 3.1 Software for graph canonization

Of the programs listed in 2.2.2, **nauty**, **Traces** and **Bliss** implement the graph canonization. Numerous experiments on them were arranged and published by Adolfo Piperno on [24]. Let us therefore reprint the ones we consider interesting.



Figure 3.1. Canonical labeling of a single graph benchmarks. Figure reprinted from [24].

Now, before applying the principles of *canonical labeling* to solve our concrete problem, let us examine the data format used in WGS (of May 2018) for the graph representation.

#### 3.2 graph6 format

Quoting the documentation on [25]: "graph6 and sparse6 are formats for storing graphs in a compact manner, using only printable ASCII characters. Files in these formats have text type and contain one line per graph".

This section will be rather detailed, so that it could be republished to introduce the users of WGS to the form of data they are facing.

### 3.2.1 Format description

Simply put, for n < 63, a graph6 string representation of a graph G, |G| = n can be obtained as follows:

- 1. Make adjacency matrix  $M = (m_{i,j})$  of G. For a simple undirected graph it is sym*metric* and it only contains zeros and ones
- 2. Construct a bitstring  $s = (m_{1,2}, m_{1,3}, m_{2,3}, m_{1,4}, m_{2,4}, m_{3,4}, \dots, m_{n-1,n}), i. e. the$ upper triangle of M without its main diagonal, read by columns
- 3. Pad s with zeros on the right to make its length a multiple of 6
- 4. Put byte  $b_1 = ||n + 63||_2$  (pad with zeros on left if necessary)
- 5. Partition s to m = |s|/6 partitions of length 6  $s_1, \ldots, s_m$
- 6. Put byte  $b_i = s_{i-1} + ||63||_2$  for  $i \in \{2, \ldots, m+1\}$
- 7. Output an ASCII interpreted string of bytes  $b_1, \ldots, b_{m+1}$

For  $62 < n < 2^{36}$  procedure works analogously, with the step 4 altered to spread n over 4 (if n < 258,047) or 8 bytes [26].

Space complexity of graph6 representation of graph G is therefore  $\Theta(|G|^2)$  where |G|denotes the order of G. See table 3.1 for practical examples.

Knowing that the G is large and sparse (i. e. its adjacency matrix is very similar to the zero matrix), sparse6 format can be used instead, in order to save space, as its space complexity is incremental to the one of the *edge list* of G rather than its adjacency matrix<sup>1</sup> [26].

### 3.2.2 Representing a graph using a graph6 format



Figure 3.2. An instance graph G to be converted in graph6 format **Example 3.1.** give a graph6 string representing graph G from figure 3.2 by following the procedure from 3.2.1:

 $<sup>^{1}</sup>$  This behavior is not implemented in WGS (as of May 2018), henceforth in this thesis the sparse6 will not be discussed as a mean of storing and comparing graphs, even though it well could be.

Graph canonization

1.

Adjacency matrix of G is 
$$M = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \end{pmatrix}$$

2. Construct s = 1101001100

3. 
$$|s| = 10$$
, therefore 2 zeros are appended.  $s = 110100110000$ 

4.  $|G| = 5, b_1 = ||68||_2 = 01000100$ 

5. Partition s to  $s_1 = 110100$  and  $s_2 = 110000$ 

```
6. Put b_2 = 110100 + 111111 = 01110011, b_3 = 110000 + 111111 = 01101111
```

7. Output string Dso (ASCII 01000100, 01110011, 01101111)

Result can be checked by e. g. using the *compute & insert* utility of WGS at http:// graphs.felk.cvut.cz/computer. In table 3.1, more examples of graphs and their graph6 representations are given.

Graph $G$	G	Adj. matrix of $G$	graph6	ASCII
	1	0	Q	01000000
12	2	$\begin{array}{ccc} 0 & 1 \\ 1 & 0 \end{array}$	<b>A</b> _	01000001 01011111
	3	$\begin{array}{ccccc} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{array}$	Bw	01000010 01110111
	6	$\begin{array}{cccccccccccccccccccccccccccccccccccc$	E?Bw	01000101 00111111 01000010 01110111

Table 3.1. Sample of simple graphs and their graph6 representation

#### 3.2.3 Parsing graph6 format

Parsing graph6 string into a graph can be easily reverse engineered from the procedure in 3.2.1, basically reversing the order of steps.

#### 3.3 **Canonized graph database**

Claim 3.1. Two simple undirected graphs are the same if and only if their graph6 representation strings equal.

**Corollary 3.2.** Two graphs  $G_1$  and  $G_2$  are *isomorphic if and only if* the graph6 representation of their canonical labels equal.

From that we can easily construct an algorithm asserting the absence of isomorphisms in a graph database D, simply by performing string comparisons of the graph6 canonical labels stored in it. Such algorithm requires the entire database and the input to be canonically labeled, but can perform as fast as in  $\Theta(log|D|)$  with the same results as the algorithm 2.2.1.

An algorithm for the graph insertion is rather trivial, utilising technologies such as an SQL unique constraint, in **PostgreSQL** implemented via a *B*-tree structure [27] sorting the **graph6** strings lexicographically, ensuring the aformentioned performance and taking up  $\Theta(|D|)$  space [28]. Implementation given in 5.1.1.

### 3.3.1 Benchmarks

For a database as big as WGS we consider crucial its ability to label a big number of graphs canonically in an efficient manner at once (as the populating of the database with a new set of graphs could become a common scenario). Since the graph storage format of the database is to be graph6 [7], bliss falls from consideration, as it works with the DIMACS format with a single graph on input.



Figure 3.3. Comparison of nauty and Traces canonical labeling algorithms.  $\mathbb{G}_i$  denotes the set of all graphs on *i* vertices (*note that the*  $|\mathbb{G}_i|$  *increases asymptotically in*  $\Omega(i!)$ , see 4.1). Full data in appendix A.1

The suggested approach to maintain a graph database D isomorphism-free is therefore by only storing canonical forms of the graphs, sorted (possibly by their graph6 string representation lexicographically) and protected from duplicates.

The canonical labeling algorithm chosen for the use in WGS is then nauty, since the *small graphs* are the core topic of WGS (as of May 2018) and on the small graphs it outperforms Traces as much as thrice (figure 3.3, table A.1). Also it stays feasible for the most of the other cases, even though its theoretical worst-case complexity is *exponential*.

# Chapter **4** Class completeness

For a graph database as big as the WGS, there is a need to know which graph classes (groups of graphs of specific order, size,...) are stored completely in it. It can then inform its users whether the data returned to their query is complete, *i. e. there exists* no graph that satisfies their query and misses in the database. It would also help its propagation, giving a catalogue of the complete collections user can browse via WGS.

This chapter gives several suggestions on how to *partially* solve these tasks in a practical instance and addresses their weaknesses, as the decisioning process is usually vulnerable to a bad quality of data, in our case mainly NULL values where some graph property is expected.

## 4.1 Loose formal definition of the problem

**Convention 4.1.** Let us relax the term graph property condition (hereinafter referred to simply as a condition) a little. It will usually be mentioned in a relation to some graph database  $D = (\mathbb{G}, f, R, m)$ , constraining some of the properties from f to yield a specific functional value for the graphs satisfying the condition. As the functions in f typically stand for some actual graph property (such as a graph order), term will be also used for graphs not included in D, having this property. To avoid misinterpretation, conditions will be always given in human language.

**Convention 4.2.** Let us call a set of *all* unlabeled (*i. e. not containing an isomorphic couple*) simple undirected graphs satisfying *every* element of some set of conditions C a **graph class** satisfying C. Graph classes are typically infinite, unless the conditions give an upper bound for the order of graphs in it.

**Problem 4.3.** Given a set of conditions C and a graph database  $D = (\mathbb{G}, f, R, m)$  that does not contain an isomorphic couple of graphs output whether the class X satisfying C is stored completely in the database D. We say that a graph class X is stored completely in a graph database  $D = (\mathbb{G}, f, R, m)$  if:

$$\forall G \in X : G \in \mathbb{G}$$

Solving problem 4.3 is trivial if X is infinite, as then there must always exist a graph in X that is absent in  $\mathbb{G}$ , which must be finite (see 1.1). Let us therefore focus just on the case where the properties give an upper bound of the graph order.

## 4.2 List of possible approaches

Naïve approach would be **enumerating** every graph from class X satisfying C (using a generator such as geng [29]) and checking its presence through *querying* it in D. That would lead to a good solution, yet perform quite poorly. The complex task being not only querying the database |X| times, but also extrapolating X from C.

Class completeness

Now, presume we have an access to an oracle  $\Omega$ , that, given a set of conditions C, outputs the size of X satisfying C in  $\Theta(1)$ :

$$\Omega(C) = |X|$$

The counting approach would then be to compute the size of set  $X_D = \mathbb{G} \cap X$  and compare it with  $\Omega(C)$ . The result will be correct, as:

$$\Omega(C) = |X_D| \iff |X| = |X_D| \iff |\{G : G \in X\}| = |\{G : G \in X \land G \in \mathbb{G}\}| \iff \\ \iff |\{G : G \in X \land G \notin \mathbb{G}\}| = 0 \iff \forall G \in X : G \in \mathbb{G} \quad (4.3)$$

Time complexity of this approach would therefore be  $\Theta(1) + T(|X_D|)$ , where  $T(|X_D|)$ denotes the complexity of counting all the graphs from D satisfying C.

In our case, presuming that all the graph properties in D are fully computed (*i. e.* no NULLs are present in D), the  $|X_D|$  can be computed using a PostgreSQL COUNT query which should perform in *linear* worst case complexity, with a further optimisation if an *index-restricted* COUNT is performed [30].

#### 4.3 **Relaxations to the counting approach**

As the **counting approach** was shown in previous section to give a correct solution in reasonable time, it would be desirable to implement a solution based on it. However, the approach relies on having an *oracle access* to the size of any graph class, which can be as hard to compute as enumerating that entire class. It also presumes no empty fields are present in the database which is very optimistic, as also graph properties can be very hard to compute, and therefore left empty, until the solution is found.

In order to give a practical solution, *relaxations* need to be introduced, as the presumptions of the pure counting approach above are unrealistic.

#### 4.3.1 Simulating the oracle access

The task to give a size of X that satisfies some conditions C is usually very complex. With that being said, there are also many applications (e. g. in chemistry, cryptography), that could benefit from the result being precomputed in a trustworthy manner. To that end, several rigorous sources were founded for storing this data, in form of an integer sequences.

The On-line Encyclopedia of Integer Sequences is the biggest rigorous source of precomputed values of integer sequences, 8,981 of them matching the keyword "graphs" (as of May 2018) [31]. The collection was started by N. J. A. Sloane in 1964 and continues to this day. There are numerous contributors expanding its content and verifying it. And, as of March  $6^{th}$  2018, there are already 6,358 works citing it [32]. This data could possibly be repurposed for the use in WGS, if the results computed using it would always be signed with a reference to the **OEIS** sequence they presume to be correct as all the decisioning faults could then be traced back to their source.

For the specific problem 4.3, **OEIS** sequences that are applicable usually give a number of all unlabeled graphs on n nodes having some property. I. e. |X| for class X that satisfies  $\{c, \forall G \in X : "G \text{ has } n \text{ vertices"} \}$  for some condition c. See table 4.1 for practical examples.

Hence, the oracle access can be simulated using **OEIS** for graph classes satisfying two conditions, one of them asserting a specific order of graph, the other asserting some other graph property. Size of graph classes satisfying more than these two conditions remains unknown and requires a more sophisticated approach to be derived.

$c [``\forall G \in X : \dots"]$	X  for $n = 1, 2, 3,$	OEIS ID
TRUE	$1, 2, 4, 11, 34, 156, 1044, 1044, 12346, 274668, 12005168, \ldots$	A000088
G is a tree	$1, 1, 1, 2, 3, 6, 11, 23, 47, 106, 235, 551, 1301, 3159, 7741, \ldots$	A000055
G is a forest	$1, 2, 3, 6, 10, 20, 37, 76, 153, 329, 710, 1601, 3658, 8599, \ldots$	A005195
G is a cactus	$1, 1, 2, 4, 9, 23, 63, 188, 596, 1979, 6804, 24118, 87379, \ldots$	A000083
G is vertex-transitive	$1, 2, 2, 4, 3, 8, 4, 14, 9, 22, 8, 74, 14, 56, 48, 286, 36, 380, 60, \ldots$	A006799
G is circulant	$1, 2, 2, 4, 3, 8, 4, 12, 8, 20, 8, 48, 14, 48, 44, 84, 36, 192, 60, \ldots$	A049287
G is Hamiltonian	$1, 0, 1, 3, 8, 48, 383, 6196, 177083, 9305118, 883156024, \ldots$	A003216
$\chi(G) = 3$	$0, 0, 1, 3, 16, 84, 579, 5721, 87381, 2104349, 78315231, \ldots$	A076279
$\omega(G) = 4$	$0, 0, 0, 1, 4, 30, 301, 4985, 142276, 7269487, 655015612, \ldots$	A052452

**Table 4.1.** Size of graph class X satisfying  $C = \{c, \ \forall G \in X : G \text{ has } n \text{ vertices}^*\}$  listed on Online Encyclopedia of Integer Sequences.  $\chi$  denotes chromatic number,  $\omega$  denotes clique number.

## 4.4 Count all – Find superclass algorithm

### 4.4.1 Subclass – superclass principles

**Convention 4.4.** Let us call class  $\hat{X}$  that contains *every* graph from class X a **superclass** of X, denoted as  $\hat{X} \supseteq X$  or  $X \subseteq \hat{X}$ .

$$\forall G \in X : G \in \hat{X}$$

**Claim 4.5.** For a class X satisfying set of conditions C, any class  $\hat{X}$  satisfying  $\hat{C} \subseteq C$  is its superclass.

**Claim 4.6.** If any class  $\hat{X}$  is *stored completely* in the graph database D, then also all classes X such that  $X \subseteq \hat{X}$  are stored completely in D

**Corollary 4.7.** Problem 4.3 can be easily solved for set of conditions C if any class  $\hat{X}$  satisfying  $\hat{C} \subseteq C$  is known to be stored completely in D

From that and from the table 4.1, we can already construct a simple two-phase algorithm, that first builds the partial oracle simulation from the data available and then substitutes the condition sets unknown by it with the others, satisfied by a superclass of the original X.

### 4.4.2 Count all

Procedure, that outputs a function mapping the graph classes of a known *size* to the *boolean* value that says whether the class is stored completely in D

Input: 1. A graph database D with all properties computed on every graph 2.  $n \in \mathbb{Z}$ 3. Set  $\mathbb{C} = \{C_1, \ldots, C_n\}$  where  $C_i$  is some set of conditions for  $i \in 1, \ldots, n$ 4. Set  $\Omega = \{\Omega_1, \dots, \Omega_n\}$  where  $\Omega_i \in \mathbb{Z}_0$  for  $i \in 1, \dots, n$ **Output** a function mem:  $\mathbb{C} \mapsto \{\mathbf{true}, \mathbf{false}\}$  such that:  $mem(C_i) =$ true iff  $X_i$  satisfying  $C_i$  is completely stored in D 1: procedure COUNTALL 2:  $i \leftarrow 1$ 3: while i < n do Compute  $|X_{D_i}| \leftarrow$  "number of all graphs in D satisfying  $C_i$ " 4: if  $|X_{D_i}| = \Omega_i$  then 5:  $m_i \leftarrow \mathbf{true}$ 6: else 7:  $m_i \leftarrow \mathbf{false}$ 8:  $i \leftarrow i + 1$ 9: **return**  $mem : C_i \to m_i$  for  $i \in \{1, \ldots, n\}$ 10:

Procedure count all 4.4.3, given sets  $\{C_1, \ldots, C_n\}$ ,  $\{\Omega_1, \ldots, \Omega_n\}$  listing e.g. the condition sets from 4.1 and their |X| values respectively outputs the class completeness knowledge represented by mem. Note that the procedure time complexity is

 $O(n) \cdot T(|X_{D_i}|) = O(|D| \cdot n)$  using PostgreSQL

and that its output can change only if the D has been altered. Therefore, it can be used as an *installation script* and called e.g. once a day.

#### 4.4.3 **Find superclass**

Algorithm, that outputs whether a class of graphs X satisfying given C is known to be completely stored in the given D

Input:

- 1. A function  $mem(\hat{C}) = true \ iff \ \hat{X} \ satisfying \ \hat{C} \ is \ completely \ stored \ in \ D$ As from the *count all* algorithm
- 2. Function conditionsThatAreImpliedBy mapping a single condition to a set of conditions.

3.  $n \in \mathbf{Z}_0$ 

4. Set of conditions  $C = \{c_1, \ldots, c_n\}$ 

**Output true** if X that satisfies C is known to be stored completely in D, false otherwise

1: procedure DoesCompletelyStoredSuperclassExist

2:	$i \leftarrow 1$
3:	$len \leftarrow n$
4:	while $i \leq len \operatorname{do}$
5:	for all $c \in conditionsThatAreImpliedBy(c_i)$ do
6:	$\mathbf{if} \ c \notin C \ \mathbf{then}$
7:	$len \leftarrow len + 1$
8:	$c_{len} \leftarrow c$
9:	$C \leftarrow \{c_1, \dots, c_{len}\}$
10:	for all $\hat{C} \subseteq C$ do
11:	$\mathbf{if} \ mem(\hat{C}) = \mathbf{true} \ \mathbf{then} \ \mathbf{return} \ \mathbf{true}$
12:	return false

Procedure find superclass, on the other hand, should be called on every user's query on D as if any well-populated (*installed*) function mem is available, it can translate various condition sets to the ones that are in the domain of mem.

A supervisor (hereinafter referred to as a "graph expert") can furthermore configure the function conditionsThatAreImpliedBy to make the function try to substitute irresolvable conditions for the ones that are known to be *completely stored* in D (for example "if all bipartite graphs on n vertices are stored, then so are all the trees on n vertices"). For that, a simplified program API would be desirable.

As menacingly as the steps 10-11 might look for the *time complexity* of the algorithm, we show them to be resolvable with a single query to the *mem* (if it is represented by an SQL table) in the documented code on the attached CD.

# Chapter **5** Implementations

As the previous chapters deal with the the theoretical aspects of the problems researched in this thesis, this chapter aims to show and explain their practical instances, on the code contributed to Web Graph Service.

The code snippets in this chapter usually implement some algorithm designed in the previous chapters and are *live* on http://graphs.felk.cvut.cz (as of May 2018). As the URL and the *user interaction* triggering the snippet will always be given, reader can see the *real-case* performance and the results himself.

## 5.1 Compute properties and insert graph

The implementation 5.1.1 parses the json GET parameter from the URL it has been called upon, such as:

```
http://graphs.felk.cvut.cz/api/graph?json={"g6":"Dso","properties":[
"arc_transitive","asymmetric","bridges","cactus","edges","nodes"]}
```

and outputs the values of the requested properties for the given graph. Furthemore, if graph is not in the database yet, it *inserts* it, with all the property values that have been computed.

The program can be accessed through a *user interface* on http://graphs.felk. cvut.cz/computer designed by Sergej Kurbanov [8].

The structure of program 5.1.1 is rather straightforward:

- 1. The core function, *computeAndInsert* [line 1], accepts the Express.js Request and Response objects [33], function *computeProperties* that encapsulates an API of an engine for computing the graph properties (SageMath supported as of 23<sup>rd</sup> May 2018) and an object db encapsulating the access to the PostgreSQL connection pool.
- 2. Properties to be computed and graph6 string g6 are parsed from the requested URL
- 3. g6 is then labeled canonically through a pipe to the configured canonical label [lines 41-43] (nauty labelg in current practice)
- 4. g6 is then looked up in the database and its properties returned. Note that if the database contains only canonically labeled graphs, then it **must always find** a stored *isomorph* of the original g6 (*query.selectGraph* returns the parametrized SELECT and prevents a possible SQL injection).
- 5. If an *isomorph* was found, its properties that are not NULL are tossed from the user's query, if no properties are then left to compute, the properties of *isomorph* are responded [line 21]. Otherwise the properties left are asked from the computation engine via *computeProperties* [line 23]
- 6. As the engine responds, its results are merged with the properties of *isomorph* and returned to the user, user is further informed, if the graph requested by him was already in the database and if yes, which values were computed on his demand (as they were set to NULL previously) [line 27]. Asynchronously, the computed values are *stored* in the db for further use [lines 26, 44-51].

```
1. function computeAndInsert(req, res, computeProperties, db) {
2. let params = JSON.parse(querystring.parse(url.parse(req.url, false).query).json)
    params.g6 = canonicalLabel(params.g6)
3.
4. if (params.g6.length === 0) {
5.
     return errorResponse(res, "Graph not readable by nauty.")
6. }
7. let select = query.selectGraph(params.g6, params.properties)
8. Promise.resolve(db.query(select.sql, select.values))
    .then((isomorphs) => \{
9.
10.
        let inDb = isomorphs.rowCount === 1
11.
        let precomputedProperties = inDb ? isomorphs.rows[0] : {}
12.
        Object.keys(precomputedProperties).forEach(property => {
13.
         if (precomputedProperties[property] !== null) {
14.
          let i = params.properties.indexOf(property)
          if (i > -1) {
15.
            params.properties.splice(i, 1)
16.
          }
17.
         }
18.
19.
        })
20.
      if (params.properties.length === 0) {
21.
        return propertiesResponse(res, params.g6, precomputedProperties, {}, inDb)
22.
       } else {
23.
        return computeProperties(computePropertiesQuery(params.g6,params.properties),
         config.timeLimit)
          .then(([status, engineOutput]) => {
24.
25.
           if (status === 200) {
26.
             storeGraph(params.g6, engineOutput, inDb, db)
27.
             return propertiesResponse(res, params.g6, precomputedProperties,
               engineOutput, inDb)
28.
            } else {
29.
             return res.status(status).json(engineOutput)
            }
30.
31.
           }).catch((err) => {
32.
            logger.info(`api/graph: ${err}`)
33.
            return errorResponse(res, "Error occurred while communicating with engine")
34.
           })
35.
       }
36. \}).catch(err => {
37. logger.error(`DB error: ${err}`)
38.
      res.status(500).json({message: "Something broke in the DB"})
39. })
40.}
41. function canonicalLabel(g6) {
42. return (spawnSync(config.canonicalLabelSpawn,{ input:
          (g6 + "\n")}).output[1] + "").split("\n")[0]
43. }
44. function storeGraph(g6, fields, inDb, db) {
45. let storeGraph = inDb ? query.updateGraph(g6, fields) : query.insertGraph(g6, fields)
46. db.query(storeGraph.sql, storeGraph.values, (err, data) => {
47. if (err) {
48.
        logger.warning(err)
49.
      }
50. })
51. }
```

**Code snippet 5.1.1.** Node.js server-side handler for the *compute & insert* section of WGS. The full code can be found at the attached CD

## 5.2 "Count all – Find superclass" graph counter

The counter and its precomputed results can be checked simply by making a query to the graph database on WGS homepage by clicking fetch and in the section Complete collections at http://graphs.felk.cvut.cz/complete respectively, the counter API can be accessed on the following URL with conditions parameter like:

```
http://graphs.felk.cvut.cz/api/checkCompleteness?conditions=[{"column":
"nodes","operator":"=","value":"10"}]
```

As the problem of deciding the *class completeness* requires a more complex microsystem, it was built as a standalone **python** package. As the COUNT queries over a large data are costly in **PostgreSQL** [30], and so is fetching tens of sequences from **OEIS**, they should not be performed *on-demand*.

For that use, another entity model, parallel to the table of graphs needs to be introduced.

G	GraphCounter
id	int
complete	bool
$\operatorname{count}$	int
possible	int
importance	int
comparators	key:value pairs
source	Source
parent	GraphCounter
children	GraphCounter
$property_1$	$type(property_1)$
:	:
$property_n$	$type(property_n)$



The fields  $property_1, \ldots, property_n$  denote the property arguments of the conditions supported by the counter (if  $property_i = NULL$ , then the condition with such an argument is satisfied by every graph). The creation of the tables is a part of the installation script, so all it takes for the graph expert to introduce a new one is to create a class extending Condition within the python package db\_counter.conditions.

The key:value pairs data type (in PostgreSQL implemented by an extension hstore) was preferred to introducing another entities, as in this case the performance (suffering from a possible additional SQL JOIN) is superior to the design niceties.

Now, to enable profits from this architecture, two separate scripts were written:

- The "count all" **installation script** that queries all the configured sequences from **OEIS** and matches their values against the numbers of graphs in database matching a single condition and having a specific order
- The "find superclass" **lookup script** that accepts a set of *conditions* C from the user and, presuming the counter to be *installed*, outputs, whether the class of graphs X *satisfying* C is known to be stored completely in the WGS or whether it is known not to be stored completely.

**Both the scripts were excluded from the text of thesis** as they basically just follow the algorithms 4.4.2 and 4.4.3 and as their logic is spread amongst multiple **Python**  classes. Their full code documented via Numpy style docstrings is appended instead. See files install.py and check.py in the folder git/db\_counter of the attached CD.

### 5.2.1 Configuring the class inclusion rules through the Python API

A graph expert can alter the decisioning logic of the find superclass algorithm simply by altering the contents of the python package db\_counter.condition.

For that use, a simple **Python API** (applicable for classes extending the **Condition** class) have been designed, to empower easy and clean maintenance of sophisticated graph class inclusion rules, such as the ones listed on **ISCGI** [35]. Two examples from the *live* program will follow.

```
1. class VertexTransitive(Condition):
2.
      """Condition wording: "G is (not) vertex transitive".
      Adds up the logic: vertex transitive => regular, irregular => not vertex transitive"""
3.
4.
     implies = [Regular]
5.
      column = "vertex_transitive"
6.
      source = Oeis("A006799")
7.
      type = BOOLEAN
8.
9. class Tree(Condition):
       """Condition wording: "G is (not) a tree".
10.
       Can be also used to extrapolate |G| if not constrained by user.""
11.
       implies = [Bipartite, Connected]
12.
13.
       column = "tree"
14.
       source = Oeis("A000055")
       type = BOOLEAN
15.
       def extrapolate_nodes(self, query):
16.
          edges = query.get(Edges)
17.
18.
          if self.value and edges is not None:
             return Nodes(edges.value + 1, edges.comparator)
19.
20.
          return super().extrapolate_nodes(query)
```

**Code snippet 5.2.2.** Example usage of the suggested **Python API** for configuring the *subclass—superclass* relations of **BOOLEAN** conditions (comments trimmed).

The code snippet 5.2.2 cofigures the graph classes *vertex-transitive* and *tree*, which are trusted to be completely stored on n nodes if the count of graphs in the database having n nodes and the property comply to the one listed in OEIS sequences A006799 and A000055 respectively.

The program will parse the following informations from the VertexTransitive class variable implies:

- 1. If all *regular* graphs are known to be completely stored in the database, then also all *vertex-transitive* graphs are known to be stored
- 2. If all *not vertex-transitive* graphs are known to be completely stored in the database, then also all *irregular* graphs are known to be stored
- 3. Condition set containing {*vertex-transitive,irregular*} is *unsatisfiable* (0 graphs are in the complete class)

Method extrapolate\_nodes of class Tree will be called if present in the user's query and if the user did not explicitly bound the graph order and it will suggest, that the bound of graph order is the bound of graph size - 1, if a bound for the graph size is

1.	class Edges(Condition):
2.	"""Condition wording: "G has m edges"
3.	Adds up the logic: If $G$ has less or equal to n vertices,
4.	it can't have more than ((n-1)*n)/2 edges i.e. such a query would be UNSATISFIABLE"""
5.	column = "edges"
6.	<pre>source = Source("Gordon Royle''s small graphs",</pre>
7.	"http://staffhome.ecm.uwa.edu.au/~00013890/remote/graphs/index.html#nums")
8.	type = INTEGER
9.	def contradicts(self, query):
10	. <b>if</b> self.comparator <b>in</b> FLOORED <b>and</b> query.nodes.comparator <b>in</b> CEILED $\setminus$
11	. <b>and</b> self.value > query.nodes.value*(query.nodes.value - 1) / 2:
12	. return True
13	. <b>return</b> super().contradicts(query)

**Code snippet 5.2.3.** Example usage of the suggested Python API for configuring the rule  $|V(G)| \cdot (|V(G)| - 1) \ge |E(G)|.$ 

known. If an equality such as graph size=4 is given, then the graph order is also bound to an equality (graph order = 3)

The code snippet 5.2.3 gives an instance of a more complicated condition, that is constraining an *integer* property values. Note that the trusted data is not an one-dimensional array as in the case of **OEIS** sequences, and so the value quantities have to be hard-coded into the class method value\_quantity (stripped from the thesis text, to be found in class db\_counter.condition.Edges)

The contradicts method will, in this case, make the *find superclass* algorithm identify a set of conditions containing a lower bound for edges higher than the n(n-1)/2on *n* vertices as unsatisfiable.

### 5.2.2 Count all — no NULLS heuristics

1. def install(self, superset_complete=False, nulls_count=None):			
2.	<pre>self.possible = self.last.value_quantity(self.last.value, self.nodes.value)</pre>		
3.	<pre>self.count = self.possible if superset_complete and nulls_count == 0 \</pre>		
4.	else SQLCounter.count_graphs(self)		
5.	self.complete = superset_complete or self.count == self.possible		
6.	if not self.complete and (		
7.	nulls_count <b>is not None and</b> self.count + nulls_count >= self.possible):		
8.	self.complete = None		
9.	<pre>self.source = self.refs(self.last.source.hstore(self.last.value))</pre>		
10.	self.importance = $4 - len(self.conditions)$		
11.	SQLCounter.insert(self)		
12.	return self.complete		

### Code snippet 5.2.4. "No NULLS" heuristics from git/db\_counter/conditions/Query.py

As the **count all** procedure is costly to perform over a large graph database, a simple heuristics, presuming the database to be *computed correctly* (where not NULL) was introduced [Line 3 of 5.2.4]. Before evaluating the completeness of any of the basic conditions on n vertices, the NULL values in the database column related to that condition are counted. Then, if *all the graphs on n vertices* are known to be stored in the database and the count of NULLs in the significant column is 0, counting the graphs matching conditions is skipped and presumed to be equal to its maximum value possible.

Note that if no NULLs are present in the column, the column is fully computed and if a superclass is known to be complete, the COUNT query must indeed output the same value.



**Figure 5.2.** Count all — "No NULLS" heuristics performance impact  $\mathbb{G}_i$  denotes the set of all graphs on *i* vertices. Full data in appendix A.2

# Chapter 5 Conclusions

Of several tested approaches to prevent **isomorphism** over the user extensible graph database, the **canonical labeling** of the database which is then constrained on the graph *uniqueness* was shown to be the *best practice*. The **on-insert isomorphism checking** was rejected with a brief argumentation of its weaknesses.

For the problem of deciding the **completeness of graph classes** stored in graph database, an ad-hoc algorithm called "count all – find superclass" was designed exploiting the introduced *graph class inclusion* principles and is yet to be challenged by other researchers as the problem it is solving is rather unique.

Several metrics have been taken on it, to show its performance and its possible enhacements achievable through a *heuristic* presuming the graph properties in database to be correctly computed (or NULL).

The entire code and related research was contributed to the Web Graph Service on http://graphs.felk.cvut.cz concurrently to the works described in [8] and [7]

For testing purposes WGS was populated with 12,293,475 *non-isomorphic* graphs, making it, with a solid probability the biggest openly queriable *graph database* (relating to the 1.2).

# 6.3 Recommendations

As an in-depth research of the problems of WGS and *graph databases* in general was an inseparable part of the thesis, we would like to conclude with several recommendations and topics that would, in our humble opinion, be worth a further research or consideration.

- If WGS is to also include large graphs, consider using sparse6 format, as the graph6 is easy to compute, but its size is incremental to the size of the graph's adjacency matrix. The size of sparse6 is, however, incremental to the length of the graph's edge list, hence it could reduce its space complexity drastically for the large *sparse* graphs [26].
- In fact, it should be considered, whether to use graph6 and sparse6 formats at all, as they deal with the non-printable ASCII characters in a rather obscure way (see 3.2.1). An initiative could be taken in offering a similar format, encoding the *adjacency matrix* or the *edge list* serialized as in 3.2.1 through a Base64 encoding [36], that is a more standardized solution of the given problem.
- In the first generation of WGS, PostgreSQL was chosen as its database engine, favoured by its developer. Yet it performs poorly when it comes to e. g. counting all the graphs stored in it, as the *visibility bit* has to be checked for every row due to its *transactional approach* [30]. Finding a way around that would be desirable, possibly by migrating the WGS to another database engine (MySQL MyIsam coming to mind when relations are not necessary).
- Making the documentation of the WGS API, mostly developed in [7], public and well accessible through the webpages (maybe even with a further propagation) could, as

we believe, be a strategic thought on how to spread thought about it, as it offers a great magnitude of an interesting data and is ready to let it be accessed *programatically*.

- The "find superclass" procedure can not extrapolate knowledge about the completeness of graph class matching an *inequality* condition (such as  $|G| \leq 5$ ) of the completnesses of *multiple* classes matching some *equality* conditions ( $|G| = 0, |G| = 1, \ldots, |G| = 5$ ). The functionality would complicate the configuration of the system from the side of a graph expert a little, but the new knowledge seems to be worth it. Possibly, such a code will be contributed to WGS as an aftermath to this thesis.
- An interesting concept introduced by ToTo Tree Width Webservice on http:// treedecompositions.com/#/twathomebatch is to employ the computing power of the client devices to solve demanding computations of graph properties. It is considerable to be integrated in WGS. However, a *user base* has to be assembled first as no clients equal no results.
- A generator such as geng [29] could be used on the client's demand to generate graphs that are absent in the database. *I. e. when the user queries a class of graphs that is not present in the database, generator tries to fix that "issue"*. This could be a major user *delighter*, giving an access to even more data than there actually *are* in the database.

## References

- Gordon Royle. Small graphs. http://staffhome.ecm.uwa.edu.au/~00013890/remote/graphs/index.html. [Online; accessed May 16<sup>th</sup> 2018].
- Brendan McKay. Combinatorial data Graphs. http://users.cecs.anu.edu.au/~bdm/data/graphs.html. [Online; accessed May 16<sup>th</sup> 2018].
- [3] Neo4j, Inc. *neo4j*. 2018.
   https://neo4j.com/. [Online; accessed May 17<sup>th</sup> 2018].
- [4] G. Brinkmann, K. Coolsaet, J. Goedgebeur, H. Mélo. House of Graphs: a database of interesting graphs, Discrete Applied Mathematics. 161:311-314, 2013. https://hog.grinvin.org/. [Online; accessed May 16<sup>th</sup> 2018].
- [5] Pisanski, T., Marušič, D., Potočnik, P., Orbanić, A., Horvat, B. & Lukšič, P. Encyclopedia of graphs. 2012—2017. http://atlas.gregas.eu. [Online; accessed May 16<sup>th</sup> 2018].
- [6] Stevan Kelk, Rim van Wersch. ToTo: An open database for computation, storage and retrieval of tree decompositions. 2016.
   http://treedecompositions.com/#/databasequery. [Online; accessed May 16<sup>th</sup> 2018].
- [7] Tomáš Roun. Graph Database Fundamental Services. Czech Technical University in Prague, 2018. [CTU Bachelor thesis].
- [8] Sergej Kurbanov. Graph Database User Interface. Czech Technical University in Prague, 2018. [CTU Bachelor thesis].
- Herbert Ullrich & RNDr. Marko Genyk-Berezovskyj. 1<sup>st</sup> generation of WGS maintenance and population tools. 2017.
   http://bertik.net/wgs\_2017.php. [Redirect shortlink for CTU GitLab; accessed May 20<sup>th</sup> 2018].
- [10] Herbert Ullrich. Graphs collection. 2017.
   http://graphs.felk.cvut.cz:8765/db/. [Online; accessed May 24<sup>th</sup> 2018].
- [11] Wikipedia contributors. Graph isomorphism Wikipedia, The Free Encyclopedia. 2018.

https://en.wikipedia.org/w/index.php?title=Graph\_isomorphism&oldid=837019374. [Online; accessed May  $16^{th}$  2018].

- [12] Robert Sedgewick. Algorithms in C. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA ©1990, 1990. ISBN 0-201-51425-7.
- [13] Laszló Babai. Graph isomorphism in quasipolynomial time. Cambridge, MA, USA, 2016. ISBN 978-1-4503-4132-5.
- [14] László Babai. Graph Isomorphism. 2017. http://people.cs.uchicago.edu/~laci/update.html. [Online; accessed May 16<sup>th</sup> 2018].

- [15] Neil J. A. Sloane. A006125. 2000.
   http://oeis.org/A006125. [Online; accessed May 17<sup>th</sup> 2018].
- [16] Neil J. A. Sloane. A000088. 2000. http://oeis.org/A000088. [Online; accessed May 17<sup>th</sup> 2018].
- [17] Jonathan L. Gross, Jay Yellen. Graph Theory and its Applications. Edition 2 edition. Chapman & Hall/CRC, 2003. ISBN 978-158488-505-4.

- [18] Brendan D. McKay. Practical Graph Isomorphism. 1981.
- [19] Brendan D. McKay, and Adolfo Piperno. Practical graph isomorphism II. Journal of Symbolic Computation. 2014, 60 (0), 94 - 112. DOI 10.1016/j.jsc.2013.09.003.
- [20] Brendan McKay, Adolfo Piperno. Search Tree Nauty Traces. 2011-2018. http://pallini.di.uniroma1.it/SearchTree.html. [Online; accessed May 22<sup>nd</sup> 2018].
- [21] P. T. Darga, K. A. Sakallah, and I. L. Markov. Saucy3. 2012. http://vlsicad.eecs.umich.edu/BK/SAUCY/. [Online; accessed May 25<sup>th</sup> 2018].
- [22] Tommi Junttila, and Petteri Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX07). 207, 135 - 149.
- [23] Wikipedia contributors. Graph canonization Wikipedia, The Free Encyclopedia. 2017. https://en.wikipedia.org/w/index.php?title=Graph\_canonization&oldid= 815257797. [Online; accessed May 24<sup>th</sup> 2018].
- [24] Brendan McKay, Adolfo Piperno. Nauty Traces. 2011-2018. http://pallini.di.uniroma1.it/. [Online; accessed May 22<sup>nd</sup> 2018].
- [25] Brendan McKay. Graph formats. https://users.cecs.anu.edu.au/~bdm/data/formats.html. [Online; accessed May 17<sup>th</sup> 2018].
- [26] Brendan McKay. Formal definition of graph6 and sparse6 formats. https://users.cecs.anu.edu.au/~bdm/data/formats.txt. [Online; accessed May 19<sup>th</sup> 2018].
- [27] PostgreSQL Contributors. PostgreSQL: Documentation 9.4: Constraints. 2017. https://www.postgresql.org/docs/9.4/static/ddl-constraints.html#DDL-CONSTRAINTS-UNIQUE-CONSTRAINTS. [Online; accessed May 18<sup>th</sup> 2018].
- [28] Bayer, R.; McCreight, E. Organization and Maintenance of Large Ordered Indexes. Acta Informatica. 1972, 173-189. [Published 1972 in Acta informatica, pages 173– 189].
- [29] Brendan McKay, Adolfo Piperno. nauty and Traces User's Guide (Version 2.6). 2016.

http://pallini.di.uniroma1.it/nug26.pdf. [Online; accessed May 19<sup>th</sup> 2018].

- [30] PostgreSQL Wiki contributors. PostgreSQL Wiki Slow Counting. 2015. https://wiki.postgresql.org/index.php?title=Slow\_Counting&oldid=26186. [Online; accessed May 20<sup>th</sup> 2018].
- [31] N. J. A. Sloane, OEIS Contributors. Sequences matching "Graph". 1964 2018. https://oeis.org/search?q=graph. [Online; accessed May 20<sup>th</sup> 2018].
- [32] OEIS Wiki Contributors. Works citing OEIS. 2018. https://oeis.org/wiki/Works\_Citing\_OEIS. [Online; accessed May 20<sup>th</sup> 2018].

[33] Node.js foundation. Request, Response - Express.js documentation. 2018. https://expressjs.com/en/api.html#req. [Online; accessed May 23<sup>rd</sup> 2018].

- [34] LLC Pony ORM. PonyORM Editor. 2018. https://editor.ponyorm.com/. [Online; accessed May 23<sup>rd</sup> 2018].
- [35] H. N. de Riddler. Information System on Graph Classes and their Inclusions (IS-GCI). 2001-2014.
   http://www.graphclasses.org/. [Online; accessed May 25<sup>th</sup> 2018].
- [36] Wikipedia contributors. Base64 Encoding Wikipedia, The Free Encyclopedia. 2018.

https://en.wikipedia.org/w/index.php?title=Base64&oldid=842752537. [Online; accessed May  $25^{th}$  2018].

# Appendix **A** Data used for visualisations

All the experiments were run on the Acer 572G (Intel Core i7-4712MQ, Samsung 750 250GB bulk SSD drive)

# **A.1** Batch canonical labeling with nauty and Traces

Measurement on  $\mathbb{G}_0$  was tossed as both the  $\mathsf{nauty}$  and  $\mathsf{Traces}$  exited with an error code.

set	set	nauty time [s]	Traces time [s]
$\mathbb{G}_0$	1	0.1200251579284668	0.10639262199401855
$\mathbb{G}_1$	1	0.0011630058288574219	0.001079559326171875
$\mathbb{G}_2$	2	0.0008225440979003906	0.0008411407470703125
$\mathbb{G}_3$	4	0.0007429122924804688	0.0008189678192138672
$\mathbb{G}_4$	11	0.0008318424224853516	0.0009133815765380859
$\mathbb{G}_5$	34	0.0008931159973144531	0.00115966796875
$\mathbb{G}_6$	156	0.0012364387512207031	0.0021255016326904297
$\mathbb{G}_7$	1044	0.0028541088104248047	0.007835626602172852
$\mathbb{G}_8$	12346	0.02155613899230957	0.05855250358581543
$\mathbb{G}_9$	274668	0.3851897716522217	1.3019285202026367
$\mathbb{G}_{10}$	12005168	15.199398756027222	53.25687122344971

# A.2 Adding heuristics to the "count all" procedure

. . .

Note that the OEIS is being accessed through its API within an instance of the procedure. That can bias the performance on small sizes of database seriously, but the main discovery of saving  $\sim 90\%$  of running time on the large data sets remains intact.

set	set	time with no NULLS heuristics $\left[ s \right]$	time without heuristics [s]
$\mathbb{G}_0$	1	13.0464260578	17.0747244358
$\mathbb{G}_1$	1	13.6582443714	12.679202795
$\mathbb{G}_2$	2	14.1572663784	12.7338917255
$\mathbb{G}_3$	4	14.3151378632	13.1491267681
$\mathbb{G}_4$	11	15.7336368561	14.4216191769
$\mathbb{G}_5$	34	16.4903476238	13.687587738
$\mathbb{G}_6$	156	17.2389614582	13.2513206005
$\mathbb{G}_7$	1044	17.6984517574	12.6007626057
$\mathbb{G}_8$	12346	14.8418450356	13.4666466713
$\mathbb{G}_9$	274668	24.8132331371	24.6525268555
$\mathbb{G}_{10}$	12005168	163.5225532055	1507.3206417561

# Appendix **B** Contents of the attached CD

The root directory of the attached CD contains the following folders:

- **git** containing the current version of the git repository at https://gitlab.fel. cvut.cz/graphs/development (as of 25<sup>th</sup> May 2018)
  - **db\_counter** containing the authored scripts for the db counter implementing "count all find superclass" documented in a **numpy** standard
- **thesis** containing this thesis in a computer-readable (*i. e. clickable*) format and Python scripts to perform the experiments
- extras
  - **orig** containing the original version of **WGS** that was designed in 2017 as the subject of Herbert Ullrich's Software–Research project
  - **on\_insert** containing the rejected on-insert isomorphism check algorithm 2.2.1
  - data containing testing sets of graphs up to 9 vertices. Also contains the SQL



- SRG Strongly regular graph
- API Application programming interface
- OEIS On-line Encyclopedia of Integer Sequences (on https://oeis.org)
- WGS Web graph service (on http://graphs.felk.cvut.cz)