

## I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Ullrich** Jméno: **Herbert** Osobní číslo: **434653**  
Fakulta/ústav: **Fakulta elektrotechnická**  
Zadávající katedra/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 isomorphism 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. Sedgwick: 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-Berezovskij, 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-Berezovskij  
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**

**F3**

**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<sup>th</sup>, 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 /

<b>1 Introduction</b> .....	1
1.1 Graph database .....	1
1.2 Examples of graph databases....	1
1.3 Web Graph Service .....	2
1.4 Motivation .....	2
<b>2 Graph isomorphism</b> .....	4
2.1 Isomorphism detection.....	4
2.2 Isomorphisms in a graph database.....	4
2.2.1 On-insert isomorphism check .....	5
2.2.2 Available software for isomorphism check .....	6
<b>3 Graph canonization</b> .....	7
3.1 Software for graph canoniza- tion .....	7
3.2 graph6 format.....	8
3.2.1 Format description.....	8
3.2.2 Representing a graph using a graph6 format .....	8
3.2.3 Parsing graph6 format.....	9
3.3 Canonized graph database.....	9
3.3.1 Benchmarks .....	10
<b>4 Class completeness</b> .....	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 access.....	12
4.4 Count all – Find superclass algorithm .....	13
4.4.1 Subclass – superclass principles .....	13
4.4.2 Count all .....	13
4.4.3 Find superclass .....	14
<b>5 Implementations</b> .....	16
5.1 Compute properties and in- sert graph .....	16
5.2 “Count all – Find super- class” graph counter .....	18
5.2.1 Configuring the class inclusion rules through the Python API .....	19
5.2.2 Count all — no NULLS heuristics .....	20
6.3 Recommendations.....	22
<b>References</b> .....	24
<b>A Data used for visualisations</b> .....	27
A.1 Batch canonical labeling with nauty and Traces .....	27
A.2 Adding heuristics to the “count all” procedure .....	28
<b>B Contents of the attached CD</b> .....	29
<b>C Glossary</b> .....	30

## Tables / Figures

<b>3.1.</b> Examples of <code>graph6</code> graph representation .....	9
<b>4.1.</b> Sizes of graph classes on $n$ vertices .....	13
<b>2.1.</b> Three drawings of graph $K_{3,3}$ ...	4
<b>3.1.</b> Canonical labeling of a single graph benchmarks.....	7
<b>3.2.</b> Example of conversion to <code>graph6</code> .....	8
<b>3.3.</b> Comparison of <code>nauty</code> and <code>Traces</code> .....	10
<b>5.1.</b> <i>Entity diagram</i> of graph WGS counter.....	18
<b>5.2.</b> 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 programmatically 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,  $f = (f_1, \dots, f_m)$  is an  $m$ -tuple of functions,  $R = (R_1, \dots, R_m)$  is an  $m$ -tuple of sets and  $f_i : \mathbb{G} \mapsto R_i$  for  $i \in \{1, \dots, m\}$ . We call  $D$  a *graph database* if:

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

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

2. For any  $G \in \mathbb{G}$  values  $f_1(G), \dots, f_m(G)$  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 programmatically (taking in account, that the adjacency matrices exceeding certain dimension are not being displayed in its interface).

<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.





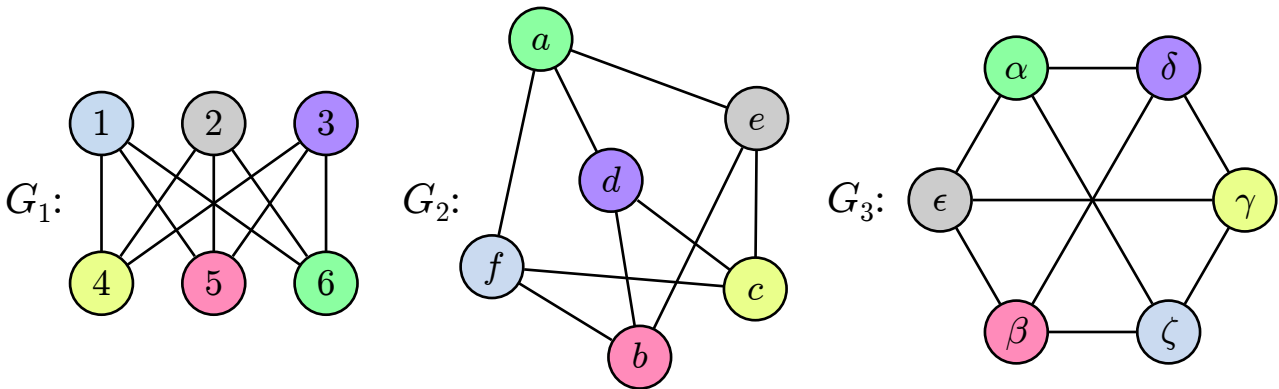
# Chapter 2

## 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  $f : G_1 \rightarrow G_2$  be a graph isomorphism and let  $v \in V_{G_1}$ . Then  $\text{deg}(f(v)) = \text{deg}(v)$  [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 ( $f_1, \dots, f_4$ ).

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  $O(|D|)$



# Chapter 3

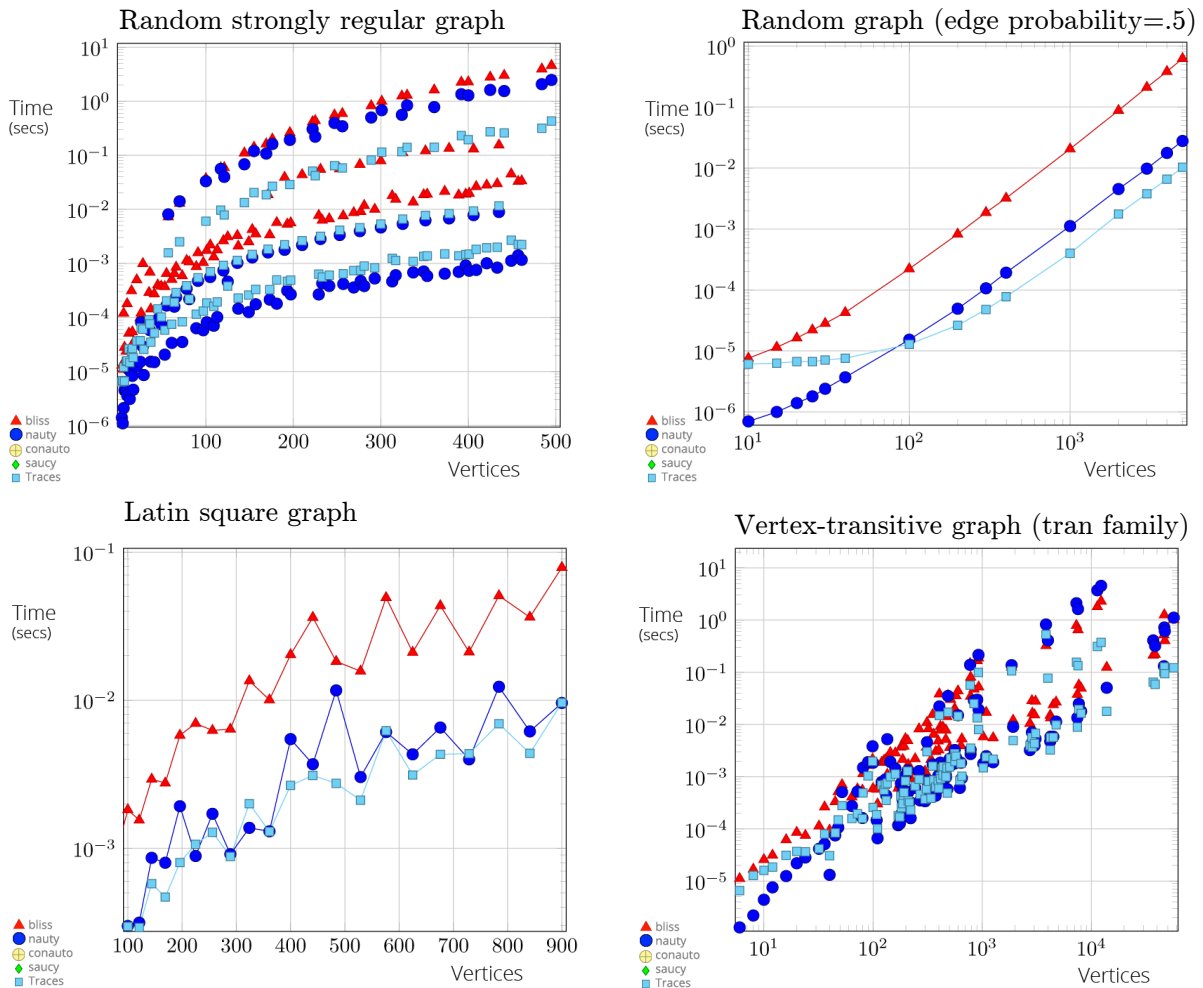
## Graph canonization

**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].





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.  $|G| = 5, b_1 = \lceil \lceil 68 \rceil_2 \rceil = 01000100$
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.



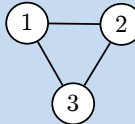
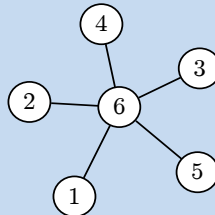
Graph $G$	$ G $	Adj. matrix of $G$	<i>graph6</i>	ASCII
	1	0	@	01000000
	2	$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$	A_	01000001 01011111
	3	$\begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$	Bw	01000010 01110111
	6	$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 \end{pmatrix}$	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.

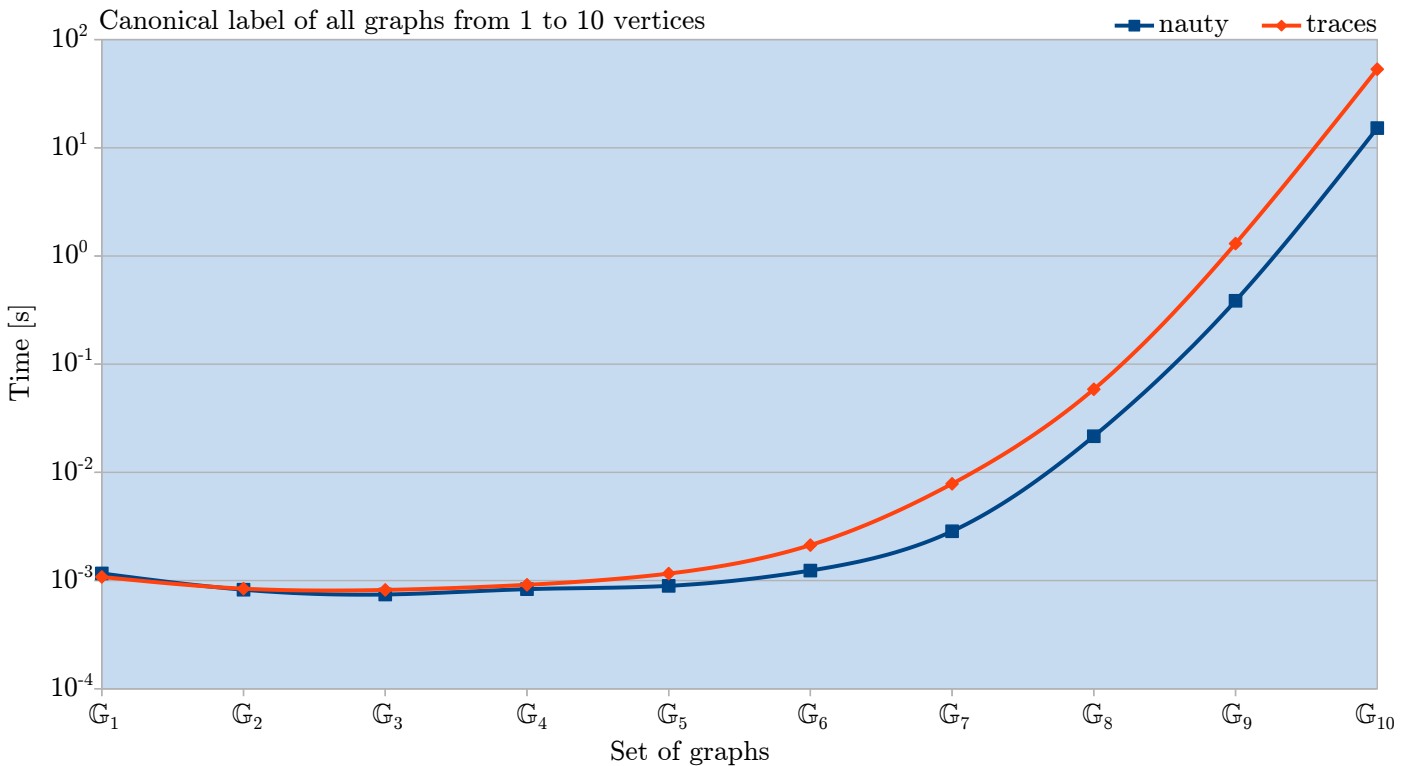
From that we can easily construct an algorithm asserting the absence of isomorphisms in a graph database  $D$ , simply by performing string comparisons of the *graph6* canonical

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 aforementioned 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.  $G_i$  denotes the set of all graphs on  $i$  vertices (note that the  $|G_i|$  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  $C$ .

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:

$$\begin{aligned} \Omega(C) = |X_D| &\iff |X| = |X_D| \iff |\{G : G \in X\}| = |\{G : G \in X \wedge G \in \mathbb{G}\}| \iff \\ &\iff |\{G : G \in X \wedge G \notin \mathbb{G}\}| = 0 \iff \forall G \in X : G \in \mathbb{G} \quad (4.3) \end{aligned}$$

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

**Table 4.1.** Size of graph class  $X$  satisfying  $C = \{c, \text{“}\forall G \in X : G \text{ has } n \text{ vertices”}\}$  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  $\mathbb{C} = \{C_1, \dots, C_n\}$  where  $C_i$  is some *set of conditions* for  $i \in 1, \dots, n$
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:  
 $mem(C_i) = \mathbf{true}$  iff  $X_i$  satisfying  $C_i$  is completely stored in  $D$

- 1: **procedure** COUNTALL
- 2:    $i \leftarrow 1$
- 3:   **while**  $i < n$  **do**
- 4:     Compute  $|X_{D_i}| \leftarrow$  “number of all graphs in  $D$  satisfying  $C_i$ ”
- 5:     **if**  $|X_{D_i}| = \Omega_i$  **then**
- 6:        $m_i \leftarrow \mathbf{true}$
- 7:     **else**
- 8:        $m_i \leftarrow \mathbf{false}$
- 9:      $i \leftarrow i + 1$
- 10:  **return**  $mem : C_i \rightarrow m_i$  for  $i \in \{1, \dots, n\}$

---

Procedure **count all** 4.4.3, given sets  $\{C_1, \dots, C_n\}$ ,  $\{\Omega_1, \dots, \Omega_n\}$  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

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

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  $mem(\hat{C}) = \mathbf{true}$  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.  $n \in \mathbb{Z}_0$
4. Set of conditions  $C = \{c_1, \dots, c_n\}$

**Output**  $\mathbf{true}$  if  $X$  that satisfies  $C$  is known to be stored completely in  $D$ ,  
 $\mathbf{false}$  otherwise

- 1: **procedure** DOESCOMPLETELYSTOREDSUPERCLASSEXIST
- 2:    $i \leftarrow 1$
- 3:    $len \leftarrow n$
- 4:   **while**  $i \leq len$  **do**
- 5:     **for all**  $c \in conditionsThatAreImpliedBy(c_i)$  **do**
- 6:       **if**  $c \notin C$  **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:     **if**  $mem(\hat{C}) = \mathbf{true}$  **then return 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 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. Furthermore, 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].





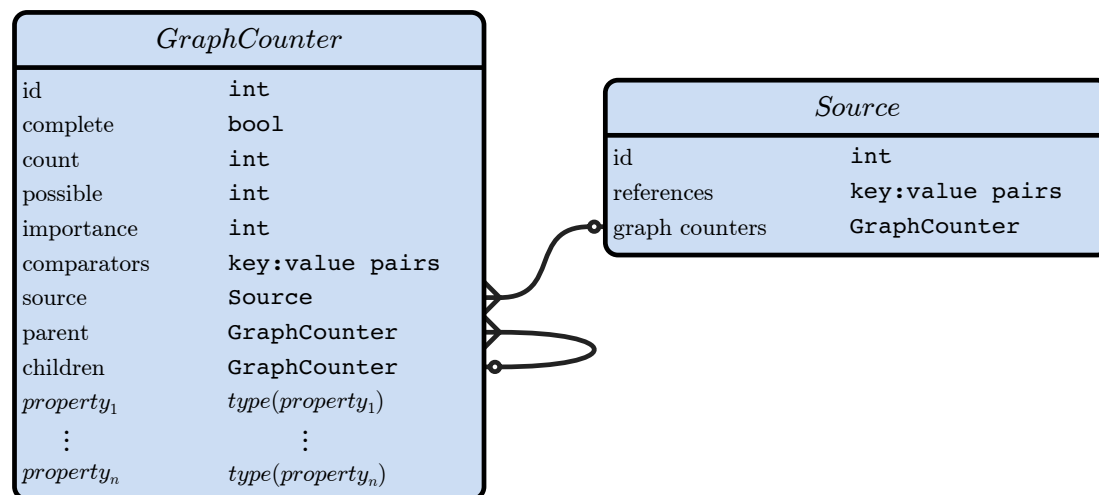
## 5.2 "Count all – Find superclass" graph counter

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.



**Figure 5.1.** Entity diagram of WGS graph counter. Created with PonyORM Editor [34]

The fields  $property_1, \dots, property_n$  denote the property arguments of the conditions supported by the counter (if  $property_i = \text{NULL}$ , 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".
3.     Adds up the logic: vertex_transitive => regular, irregular => not vertex transitive"""
4.     implies = [Regular]
5.     column = "vertex_transitive"
6.     source = Oeis("A006799")
7.     type = BOOLEAN
8.
9. class Tree(Condition):
10.    """Condition wording: "G is (not) a tree".
11.    Can be also used to extrapolate |G| if not constrained by user."""
12.    implies = [Bipartite, Connected]
13.    column = "tree"
14.    source = Oeis("A000055")
15.    type = BOOLEAN
16.    def extrapolate_nodes(self, query):
17.        edges = query.get(Edges)
18.        if self.value and edges is not None:
19.            return Nodes(edges.value + 1, edges.comparator)
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 configures 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.     source = Source("Gordon Royle's small graphs",
7.                    "http://staffhome.ecm.uwa.edu.au/~00013890/remote/graphs/index.html#nums")
8.     type = INTEGER
9.     def contradicts(self, query):
10.        if self.comparator in FLOORED and query.nodes.comparator in CEILED \
11.           and self.value > query.nodes.value*(query.nodes.value - 1) / 2:
12.            return True
13.        return super().contradicts(query)

```

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

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  $n(n - 1)/2$  on  $n$  vertices as unsatisfiable.

## ■ 5.2.2 Count all — no NULLS heuristics

```

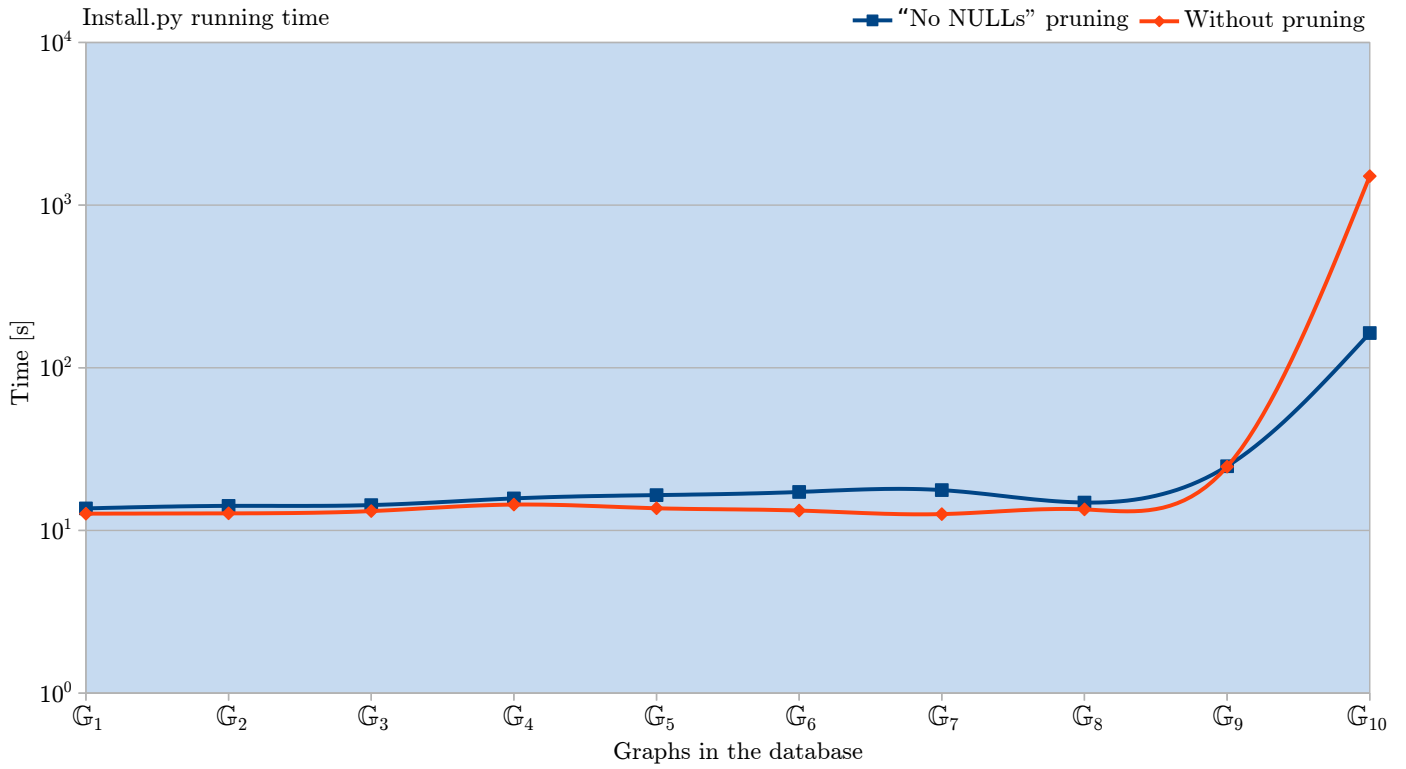
1. def install(self, superset_complete=False, nulls_count=None):
2.     self.possible = self.last.value_quantity(self.last.value, self.nodes.value)
3.     self.count = self.possible if superset_complete and nulls_count == 0 \
4.         else SQLCounter.count_graphs(self)
5.     self.complete = superset_complete or self.count == self.possible
6.     if not self.complete and (
7.         nulls_count is not None and self.count + nulls_count >= self.possible):
8.         self.complete = None
9.     self.source = self.refs(self.last.source.hstore(self.last.value))
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  $G_i$  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 enhancements 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



## References

- [1] Gordon Royle. *Small graphs*.  
<http://staffhome.ecm.uwa.edu.au/~00013890/remote/graphs/index.html>. [Online; accessed May 16<sup>th</sup> 2018].
- [2] 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].
- [9] Herbert Ullrich & RNDr. Marko Genyk-Berezovskyj. *1<sup>st</sup> generation of WGS maintenance and population tools*. 2017.  
[http://bertik.net/wgs\\_2017.php](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](https://en.wikipedia.org/w/index.php?title=Graph_isomorphism&oldid=837019374). [Online; accessed May 16<sup>th</sup> 2018].
- [12] Robert Sedgewick. *Algorithms in C*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA ©1990, 1990. ISBN 0-201-51425-7.
- [13] Laszlo 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](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](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](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<sup>th</sup> 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  $G_0$  was tossed as both the **nauty** and **Traces** exited with an error code.

$set$	$ set $	nauty time [s]	Traces time [s]
$G_0$	1	0.1200251579284668	0.10639262199401855
$G_1$	1	0.0011630058288574219	0.001079559326171875
$G_2$	2	0.0008225440979003906	0.0008411407470703125
$G_3$	4	0.0007429122924804688	0.0008189678192138672
$G_4$	11	0.0008318424224853516	0.0009133815765380859
$G_5$	34	0.0008931159973144531	0.00115966796875
$G_6$	156	0.0012364387512207031	0.0021255016326904297
$G_7$	1044	0.0028541088104248047	0.007835626602172852
$G_8$	12346	0.02155613899230957	0.05855250358581543
$G_9$	274668	0.3851897716522217	1.3019285202026367
$G_{10}$	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 [s]	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**



## Appendix C

### Glossary

- 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>)